510.84 Math. 8
Il6r
no 1022
cop. 2

Report No. UIUCDCS-R-80-1022 UILU-ENG 80 1720

SPEEDING UP COMPUTATION IN TWO-DIMENSIONAL ITERATIVE ARRAYS

by

Peter Gerald Rose

May 1980 NSF-OCA-MCS80-01561-000048

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-80-1022

SPEEDING UP COMPUTATION IN TWO-DIMENSIONAL ITERATIVE ARRAYS

by

Peter Gerald Rose

May 1980

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF-MCS80-01561 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, May 1980.

ACKNOWLEDGMENT

I would like to thank my thesis advisor, Professor Daniel Gajski, for his patient support, for his helpful suggestions, and most of all for his willingness to let me pursue my own ideas wherever they led.

[illegible]

[illegible]

NOTATIONAL CONVENTIONS USED IN THIS THESIS

Symbol	Meaning
log	logarithm with base 2
$\Longrightarrow$	"implies"
$\Longleftrightarrow$	"if and only if"
$\equiv$	"is defined to be"
$\lfloor x \rfloor$	greatest integer $\leq x$

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
OVERVIEW

1.1. Background

Two-dimensional arrays of switching circuits have been studied for some time, beginning notably with the work of Hennie.[1] Other early research in the area is reviewed in the survey article by Minnick.[2] Important contributions were later made by Kautz,[3] Akers,[4], and several others. Most of the work cited above consists of theoretical investigations into the ability of two-dimensional logic networks tc realize arbitrary combinational functions.

Of more practical interest have been the many schemes proposed which use two-dimensional iterative arrays to implement specific computations. As examples, networks have been designed for performing multiplication,[5] division,[6] and square root extraction.[7] Because of the modularity inherent in array structures, such circuits possess great advantages in the modern world of LSI technology.

This thesis explores the idea of transforming a two-dimensional array into an equivalent one which performs the same computation but in less time, preferably using no more hardware than the original array. We could thus obtain a speed increase while preserving the modularity of cellular design. Ideally, we would like to have an algorithm which automatically performs this transformation for a large class of two-dimensional networks.

1.2. Method 1

We shall consider here a rectangular network of the kind shown in Figure 1. All modules, or "cells", in the network are identical and

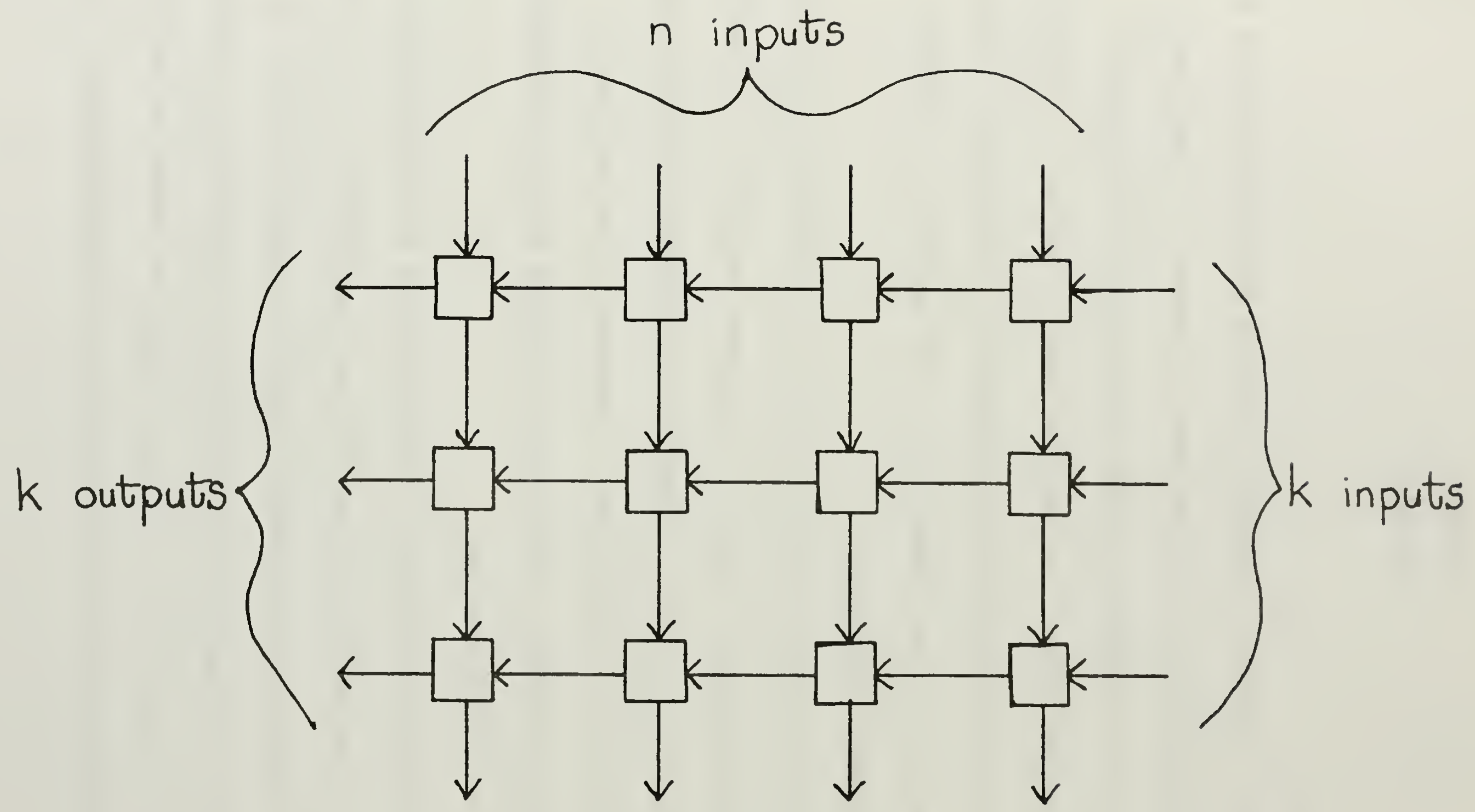

Figure 1. An (n,k) rectangular network.

consist of combinational logic only. Network inputs and outputs are as shown. Throughout this paper we shall ignore the cell outputs along the bottom edge of the network.

The array has n columns and k rows of modules and will be referred to as an (n,k) network. For convenience we shall assume throughout this work that n is a power of 2 and that $1 \leq k \leq n$. Each arrow in Figure 1 represents a bundle of m bit lines ($m \geq 1$). By S we shall denote the set of 2^m distinct states such a bundle can assume. However, we will normally think of a bundle as a single "line" carrying a signal which represents an element of S.

Figure 2 shows a single module. We shall assume that the propagation time to either output is exactly one "unit delay". The output values will be denoted as shown; we think of # and * as binary operations defined on the set S (that is, as mappings from the Cartesian product $S \times S$ into S).

We are concerned mainly with the propagation delay of the network in Figure 1, which we shall henceforth call a Method 1 network. It is easy to see that once the inputs are applied it takes $n + k - 1$ unit delays before all the outputs are valid. We want to transform this array into a faster one which produces the same outputs.

1.3. Proposed New Methods

In Chapter 2 such a transformation is described. We shall call this new way of doing the computation Method 2, Method 1 referring to the original rectangular network. Method 2 can often yield a striking improvement in performance, and it has the advantage of being applicable to any array, regardless of the operations # and *. However, Method 2 is faster than Method 1 only when k is quite small compared to n; otherwise, it is

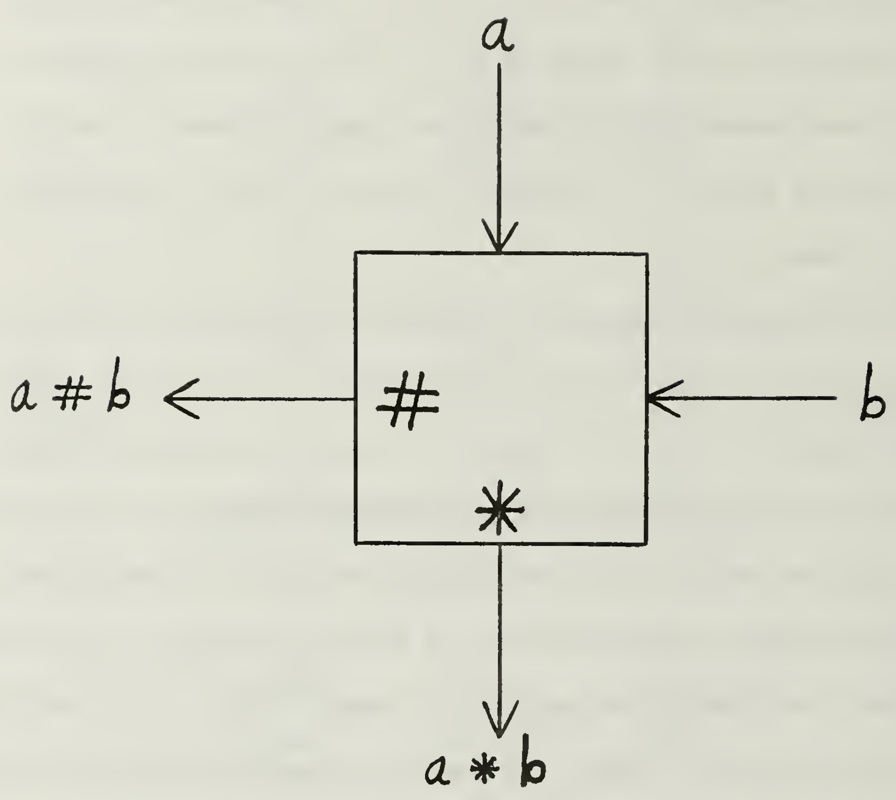

Figure 2. Single cell from rectangular network.

actually slower. Method 2 also requires more hardware.

In the search for a better transformation, Method 3 was developed; it is introduced in Chapter 3. Most of this thesis is devoted to investigating the properties of Method 3, which is a superior technique in several ways.

First of all, Method 3 is a purely geometric transformation of the rectangular array. That is, the Method 3 network uses not only the same type of module as Method 1, but also the same number of them. Only the interconnection pattern is different. Yet we shall show that for most values of k Method 3 is substantially faster than Method 1 and that in no case is it slower. Moreover, Method 3 always outperforms Method 2. Chapter 5 develops and discusses these speed comparisons.

Unfortunately, Method 3 will not work for all arrays, since in general it is not an output-preserving transformation. Only for certain instances of the cell operations # and * will the Method 1 network and the corresponding Method 3 network perform the same computation; this matter is discussed in Chapter 4. Perhaps future research will be able to modify Method 3 so as to extend its range of application. In any event, the many advantages of Method 3 give it excellent potential as a design algorithm for speeding up computations done by two-dimensional arrays.

CHAPTER 2
METHOD 2

2.1. Construction

To help explain Method 2, we shall use as an example a (4,3) rectangular network, which Figure 3 shows in detail. Let us first consider the top row of cells. Viewed by itself, it is a one-dimensional iterative array, otherwise known as a (first-order) recurrence network. There exists a general method for speeding up such a computation; it involves using semigroups of functions to obtain a tree implementation of the network. This technique has been described recently by several authors, including Gajski[8] and Unger.[9]

However, to simplify our discussion let us assume that the operation # is associative. In that case we can derive a tree implementation directly, for $y_1 = a_{14} \# (a_{13} \# (a_{12} \# (a_{11} \# x_1)))$ can be computed as $a_{14} \# ((a_{13} \# a_{12}) \# (a_{11} \# x_1))$. In this way some parallelism can be introduced into an originally serial computation. For a detailed discussion of tree-height reduction by associativity, see pp. 101 - 102 of Kuck.[10]

Anyway, Method 2 uses this technique to transform the first row of the rectangular array into the network shown in Figure 4. Figure 5 depicts the two modules used in this network; we assume each has propagation time of one unit delay. Observe that the cell in Figure 2 is simply these two modules combined into one box. With all of these modules note carefully the order of the inputs, for # and * are not necessarily commutative.

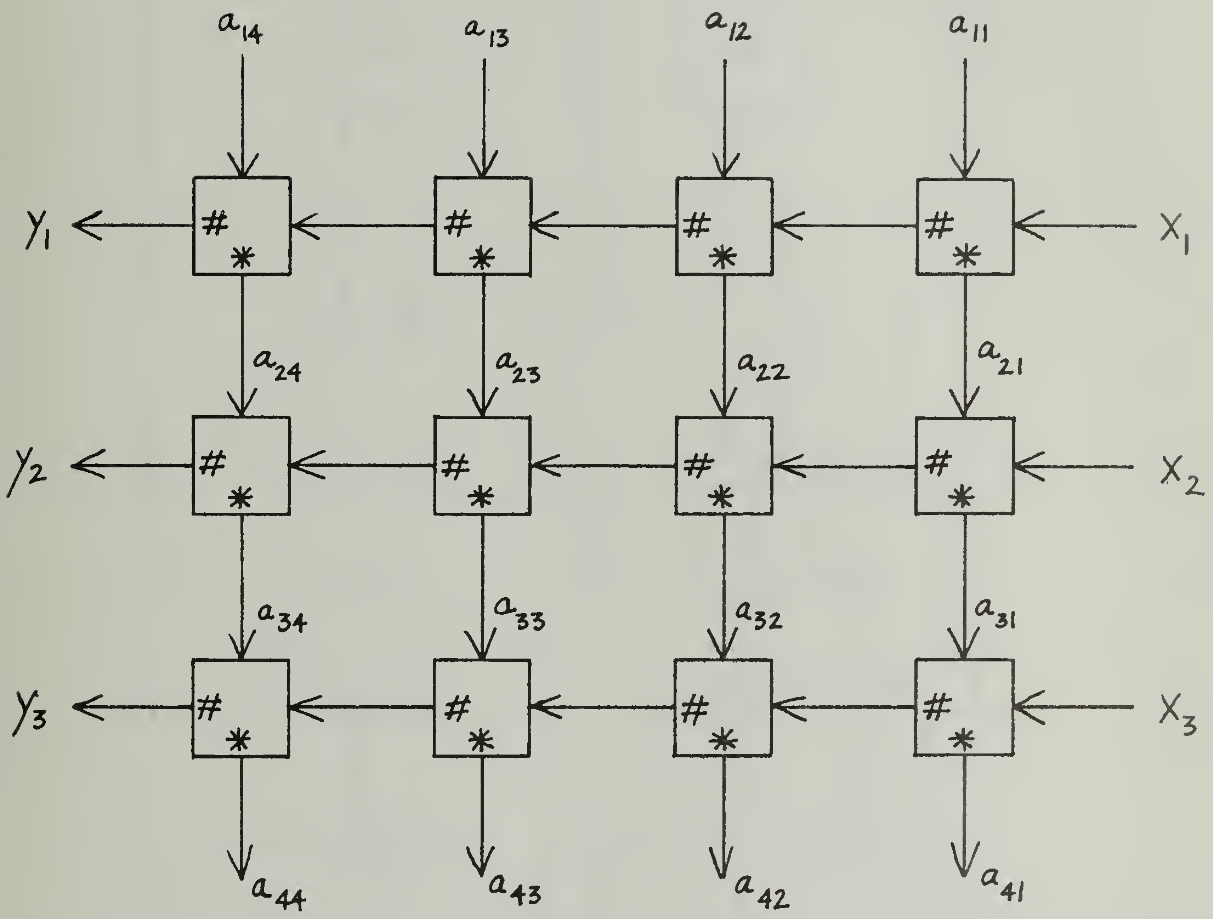

Figure 3. A (4,3) rectangular (Method 1) network.

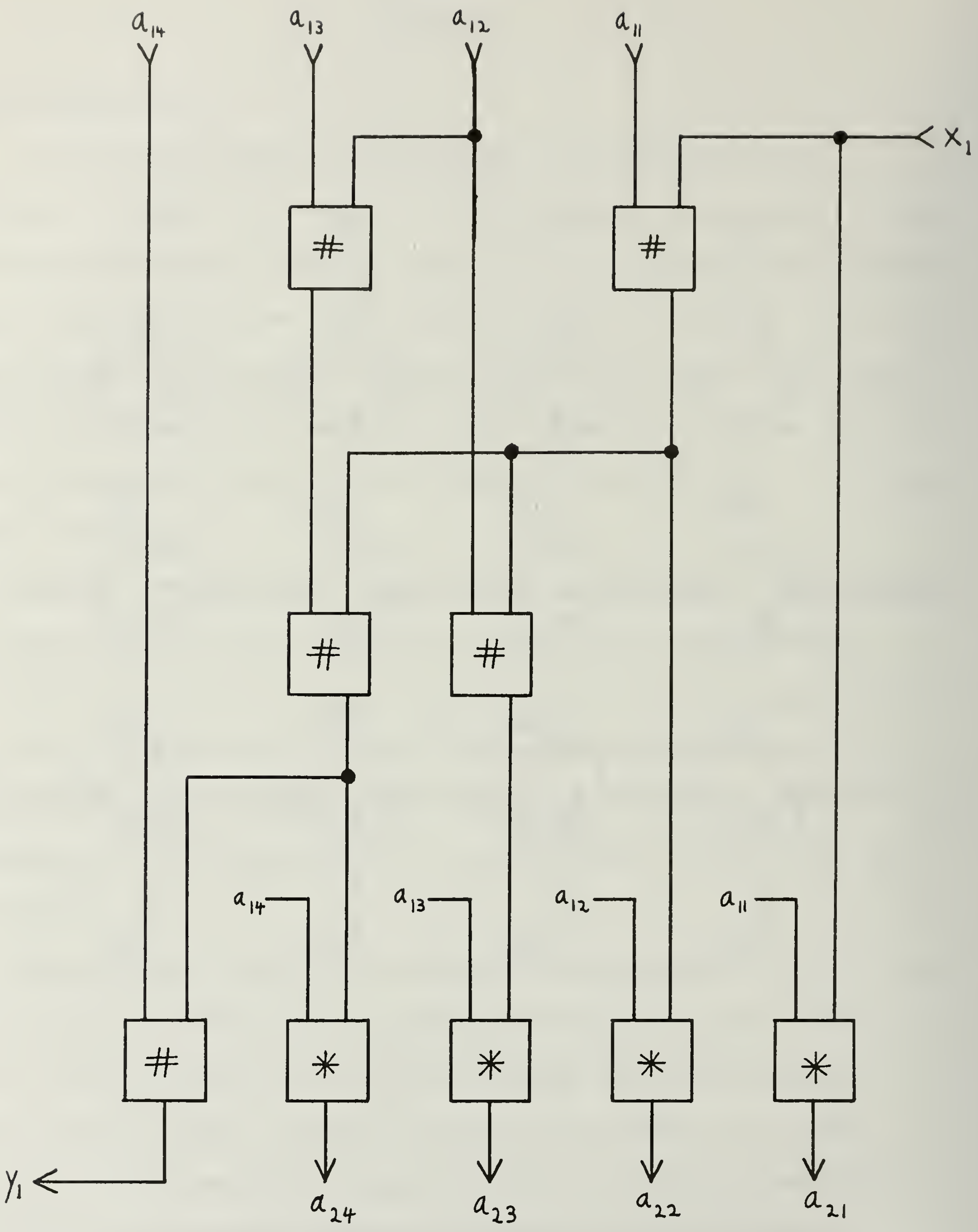

Figure 4. First stage of (4,3) Method 2 network.

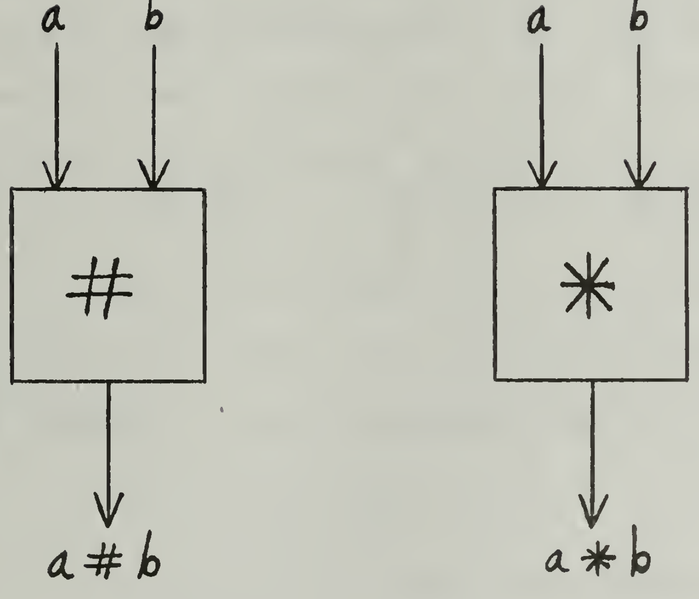

Figure 5. Modules used in Method 2 network.

It is easy to verify that the network of Figure 4 correctly computes y_1 and the a_{2i} from the inputs x_1 and the a_{1i} $(1 \leq i \leq 4)$. We now apply exactly the same transformation to the second row of the Method 1 network to obtain the second stage of the Method 2 network. This stage takes as inputs the a_{2i} from the first stage, along with x_2; it computes y_2 and the a_{3i}. The third row of the rectangular array is similarly transformed to complete the (4,3) Method 2 network. (Actually, this last stage can be somewhat simplified because the a_{4i} need not be computed.)

As another example, Figure 6 depicts one stage of a Method 2 network for $n = 16$. Here solid circles represent #-modules and open circles represent *-modules. (In this more schematic drawing, suggested by Gajski[11], one can begin to see the recursive nature of this design. That is, the circuit for n is constructed in a regular way from two circuits for $\frac{n}{2}$. For more details on this matter, see the paper by Ladner and Fischer[12], especially their Figure 3.) Again, it is easy to see that this circuit performs the desired computation. A complete (16,k) Method 2 network consists of k of these stages cascaded together.

2.2. A Bound for Computation Time

This transformation works for any (n,k) rectangular network. Basically, what Method 2 does is compute each output y_j by fanning in $n + 1$ lines with #-modules as fast as possible. Recalling that n is a power of 2, we see that each stage of the Method 2 network can complete all of its computations in $1 + \log n$ unit delays, once all of the stage inputs have become available. Thus the total propagation time for the entire (n,k) Method 2 network is no greater than $k(1 + \log n)$ unit delays,

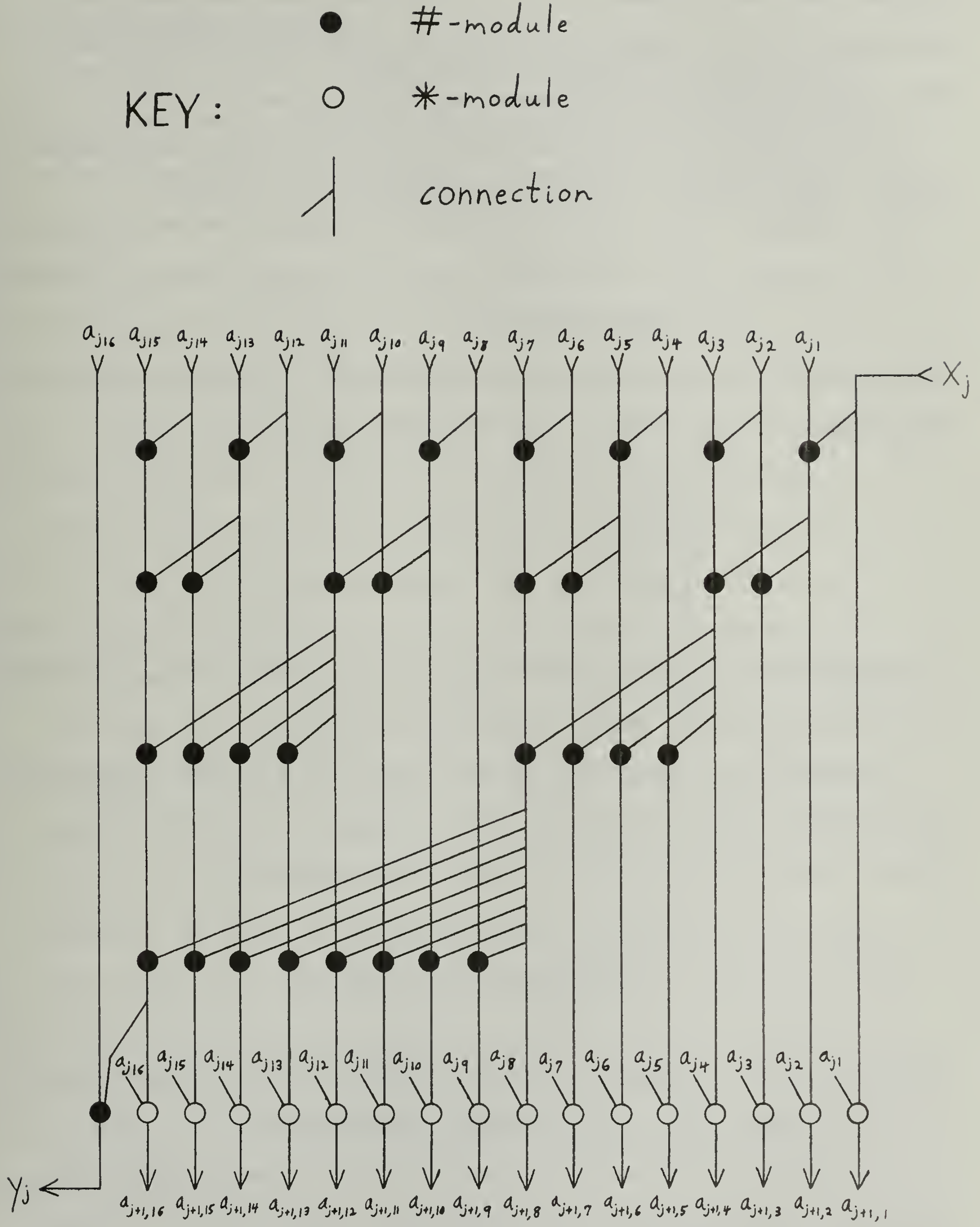
KEY:
● #-module
○ *-module
connection
a_{j16} a_{j15} a_{j14} a_{j13} a_{j12} a_{j11} a_{j10} a_{j9} a_{j8} a_{j7} a_{j6} a_{j5} a_{j4} a_{j3} a_{j2} a_{j1}
X_j
a_{j16} a_{j15} a_{j14} a_{j13} a_{j12} a_{j11} a_{j10} a_{j9} a_{j8} a_{j7} a_{j6} a_{j5} a_{j4} a_{j3} a_{j2} a_{j1}
Y_j
$a_{j+1,16}$ $a_{j+1,15}$ $a_{j+1,14}$ $a_{j+1,13}$ $a_{j+1,12}$ $a_{j+1,11}$ $a_{j+1,10}$ $a_{j+1,9}$ $a_{j+1,8}$ $a_{j+1,7}$ $a_{j+1,6}$ $a_{j+1,5}$ $a_{j+1,4}$ $a_{j+1,3}$ $a_{j+1,2}$ $a_{j+1,1}$

Figure 6. Stage j of Method 2 network for n = 16.

because there are k stages in all. We shall see later that the actual propagation time is somewhat less than this upper bound, although over the range of interest the difference is not very great.

Let T_1 denote the computation time for an (n,k) network using Method 1 and T_2 denote the time required using Method 2. We then have:

$$T_1 = n + k - 1$$
$$T_2 \le k(1 + \log n) \equiv \hat{T}_2$$

For the moment let us assume that the upper bound $\hat{T}_2$ is the true Method 2 delay time. It is trivial to prove the following result:

$$k < \frac{n-1}{\log n} \implies \hat{T}_2 < T_1$$
$$k > \frac{n-1}{\log n} \implies \hat{T}_2 > T_1$$

We shall denote the crossover point $\frac{n-1}{\log n}$ by $\hat{c}_n$. Note that $\hat{c}_n$ is generally much less than n; for example, if $n = 128$, $\hat{c}_n \approx 18$.

Thus Method 2 is faster than Method 1 only when k is small relative to n; otherwise, Method 1 is faster. If $k < \hat{c}_n$, then $k << n$, whence the speedup (as defined by Kuck[10]) of Method 2 over Method 1 is:

$$\frac{T_1}{\hat{T}_2} = \frac{n + k - 1}{k(1 + \log n)} \approx \frac{1}{k}\left(\frac{n}{\log n}\right)$$

When $k = 1$, which corresponds to the rectangular array having only one row, this becomes $\frac{n}{\log n}$, the well-known speedup obtained when a one-dimensional iterative array is replaced by a binary tree implementation. Hence, one can view $\frac{1}{k}$ as an attenuation factor which arises in the two-dimensional case.

2.3. Actual Computation Time

As mentioned earlier, the true Method 2 propagation time T_2 is in general somewhat smaller than $\hat{T}_2 = k(1 + \log n)$. To see why this is so, let us look again at the example shown in Figure 6. Although for the sake of regularity we have arranged the diagram to show 5 rows of modules, the rightmost 8 columns actually have depth less than 5. Thus (taking $j = 1$) the first stage of the Method 2 network produces output y_1 after 5 unit delays, but $a_{21}, a_{22}, a_{23}, \ldots, a_{28}$ are ready earlier. Hence, the second stage of the network can begin computing sooner than we might have expected, and many of <u>its</u> outputs are ready after far less than 10 unit delays.

Continuing down through later stages, this speedup effect begins to snowball, aided by the fact that most of the columns (even on the left side) do not have modules in every row. The effect drifts from right to left, until after a few stages the speedup starts to appear in the outputs y_j. This means that effectively the propagation time through each of the later stages is less than 5 unit delays. Hence, for the (16,k) network we find that $T_2 < \hat{T}_2 = 5k$, at least for the larger values of k. Table 1 details the situation for this example.

Naturally, the preceding discussion applies regardless of the value of n. In all cases, the effective propagation time is $1 + \log n$ only for the first several (in fact, $\log n$) stages; thereafter, the stage delay decreases. Thus, keeping n fixed, the difference between $\hat{T}_2$ and T_2 grows with increasing k. As Table 1 shows, this difference can become quite large as k approaches n.

However, we are not really interested in the larger values of k if in those cases Method 2 is outperformed by Method 1, anyway. What we would

k	$\hat{T}_2$	T_2	effective delay time of kth stage	$\hat{T}_2 - T_2$
1	5	5	5	0
2	10	10	5	0
3	15	15	5	0
4	20	20	5	0
5	25	24	4	1
6	30	28	4	2
7	35	32	4	3
8	40	35	3	5
9	45	38	3	7
10	50	40	2	10
11	55	42	2	13
12	60	44	2	16
13	65	45	1	20
14	70	46	1	24
15	75	47	1	28
16	80	48	1	32

Table 1. Propagation time (in unit delays) for (16,k) Method 2 network.

like to know is, for each n, the largest value of k for which Method 2 is faster than Method 1. Earlier we calculated this crossover point to be $\hat{c}_n = \frac{n-1}{\log n}$. (Since $\hat{c}_n$ is not in general an integer, the value of k we desire would be $\lfloor \hat{c}_n \rfloor$.) But recall that this was computed on the basis of $\hat{T}_2$, not T_2. Thus we now define c_n to be the <u>true</u> crossover point between the two methods. That is, c_n is the integer such that:

$$k \le c_n \implies T_2 < T_1$$

$$k > c_n \implies T_2 \ge T_1$$

Since in general $T_2 < \hat{T}_2$, Method 2 should compare more favorably to Method 1 than we had thought. In particular, we would expect that $c_n > \lfloor \hat{c}_n \rfloor$. But what is the magnitude of this improvement? The answer was obtained from a computer simulation of Method 2 which determined T_2 for values of n up through 1024 and all relevant values of k. From this were derived values of c_n, which are displayed in Table 2.

Note that over the range considered $\frac{c_n}{\lfloor \hat{c}_n \rfloor} < 1.2$, an improvement of less than 20%. More significantly, $\frac{c_n - \lfloor \hat{c}_n \rfloor}{n} \le .02$; that is, the difference between the true crossover and the estimated one is no more than 2% of n. In addition, the computer study showed that for $k \le c_n$ (the range over which Method 2 would be used) the difference between T_2 and $\hat{T}_2$ is not very great; the improvement is never more than 14%, and usually it is much less. T_2 and $\hat{T}_2$ are especially close when k is very small, which is precisely when we would be most likely to use Method 2 (because in this range Method 2 outperforms Method 1 by the greatest amount). Later in this paper Figure 8 and Figures 16 through 19 provide graphical comparisons

n	4	8	16	32	64	128	256	512	1024
c_n	1	2	3	6	11	19	36	66	120
$\lfloor \hat{c}_n \rfloor$	1	2	3	6	10	18	31	56	102
$\frac{c_n}{n}$	.25	.25	.19	.19	.17	.15	.14	.13	.12
$\frac{\lfloor \hat{c}_n \rfloor}{n}$	.25	.25	.19	.19	.16	.14	.12	.11	.10
$\frac{c_n}{\lfloor \hat{c}_n \rfloor}$	1	1	1	1	1.10	1.06	1.16	1.18	1.18
$\frac{c_n - \lfloor \hat{c}_n \rfloor}{n}$	0	0	0	0	.016	.008	.020	.020	.018

Table 2. Comparison between c_n and $\hat{c}_n$.

of T_1, T_2, and $\hat{T}_2$.

In conclusion, we see that although Method 2 performs better than our upper bound $\hat{T}_2$ would indicate, the difference is not particularly significant. For most practical purposes, the propagation time through an (n,k) Method 2 network can be roughly approximated by the simple formula $k(1 + \log n)$. Moreover, $\frac{n - 1}{\log n}$ serves as a reasonable estimate for the crossover point between Method 2 and Method 1.

2.4. Required Hardware

Looking at Figures 4 and 6, we see that a single stage of an (n,k) Method 2 network will require exactly n *-modules. Moreoever, it is not hard to show that the number of #-modules needed is $1 + \frac{1}{2} n \log n$. Since there are k stages, the entire Method 2 network requires nk *-modules and $nk(\frac{1}{2} \log n) + k$ #-modules. (We neglect the fact that the last stage can be simplified.)

Of course, Method 1 uses exactly nk cells. These are equivalent to nk *-modules and nk #-modules, a significant savings in #-modules over Method 2. Moreover, the Method 1 modules may possibly be simplified by gate-sharing, since two outputs are being produced from the same set of inputs. (For a discussion about design of such multiple-output circuits, see, for example, pp. 131-139 of McCluskey.[13]). Thus, from the point of view of pure switching theory, we see that Method 2 will require somewhat more hardware than Method 1.

In reality, this difference may be magnified by some pragmatic factors favoring Method 1. If we assume one cell per chip, the total package count is nk for Method 1 versus $(1 + \frac{1}{2} \log n)nk + k$ for Method 2. This is a quite relevant comparison even though the Method 2 cells are simpler, for these days the number of chips mounted on the PC boards usually has a greater impact on cost than does on-chip complexity. In addition, Method 1 uses only one type of chip, whereas Method 2 requires two types. This disparity is quite meaningful, considering the high initial cost of integrated circuit fabrication.

2.5. Fan-out and Method 2'

Another problem with Method 2 is the matter of fan-out. Looking at Figure 6, consider the lowest #-module in the column with input a_{j7}. It is required to drive 8 #-modules plus 1 *-module. Assuming that each module, # or *, presents a single "unit load", the #-module in question must be able to handle a fan-out of 9. This is the highest fan-out in the network; it is much lower for most other modules. However, since all #-modules are presumed to be identical, they must be designed to accomodate the worst case. Generalizing, it is easy to see that the maximum fan-out occurring in an (n,k) Method 2 network is $\frac{n}{2} + 1$. If n is at all large, this power requirement may well be unacceptable.

To remedy this situation we shall now introduce a variation on Method 2, called Method 2', which is based on a suggestion made by Gajski.[11] Figure 7 illustrates the new method for the case n = 16 and is to be compared to Figure 6. Both circuits produce the same outputs, a result which follows from our assumption that the # operation is associative. Moreover, in Method 2' we cascade stages exactly as we did in Method 2. What we have done in Figure 7 is to insert some additional #-modules and make the interconnection pattern more uniform.

As a consequence, fan-out in Method 2' is more equally distributed

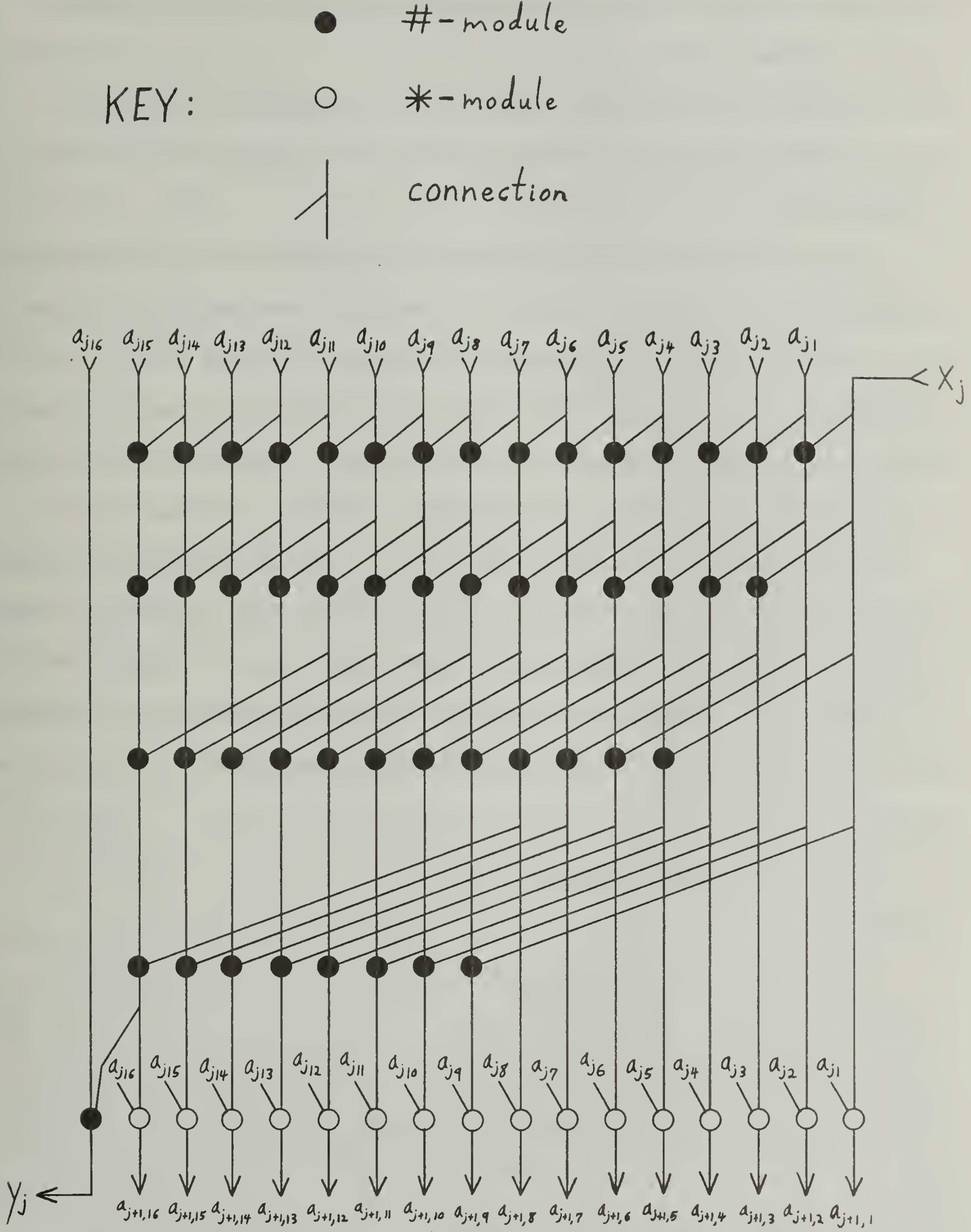
KEY:
● #-module
○ *-module
connection
a_{j16} a_{j15} a_{j14} a_{j13} a_{j12} a_{j11} a_{j10} a_{j9} a_{j8} a_{j7} a_{j6} a_{j5} a_{j4} a_{j3} a_{j2} a_{j1}
X_j
Y_j
$a_{j+1,16}$ $a_{j+1,15}$ $a_{j+1,14}$ $a_{j+1,13}$ $a_{j+1,12}$ $a_{j+1,11}$ $a_{j+1,10}$ $a_{j+1,9}$ $a_{j+1,8}$ $a_{j+1,7}$ $a_{j+1,6}$ $a_{j+1,5}$ $a_{j+1,4}$ $a_{j+1,3}$ $a_{j+1,2}$ $a_{j+1,1}$

Figure 7. Stage j of Method 2' network for n = 16.

among the #-modules. Indeed, the largest fan-out in Figure 7, handled by the rightmost module in the top row, is only 4 unit loads. In general one can see that the maximum fan-out in an (n,k) Method 2' network is log n. This is well within the limits of current technology, even for very large n.

However, Method 2' does entail some disadvantages. The most obvious drawback is that of extra hardware. A straightforward calculation shows that the number of #-modules required per stage for Method 2' is $n(\log n - 1) + 2$, compared to $\frac{1}{2} n \log n + 1$ for Method 2. Hence, the new method uses approximately twice as many #-modules. (Note that the number of *-modules is the same for both methods: exactly n per stage.)

Another penalty incurred with Method 2' is a slight decrease in speed. Comparing Figures 6 and 7, we see that propagation times to corresponding outputs are identical (assuming all inputs arrive simultaneously.) Yet when many stages are cascaded together, the overall network delay times for the two methods differ! This surprising result stems from the fact that Method 2' fills in most of the gaps in the #-module "matrix" of Figure 6. This hinders the stage-by-stage speedup effect described earlier for Method 2.

However, some of this speedup does occur in Method 2', developing roughly as follows. Referring to Figure 7, we see that the right half of the circuit computes faster than $1 + \log n$, the delay time for the left half. Now note that the outputs of each stage are fed into the inputs of the next stage shifted to the left by one column. From this it follows that the speedup effect travels to the left by one column per stage; it begins to appear at the network outputs y_j only after the first $\frac{n}{2}$ stages.

Thus for the first $\frac{n}{2}$ stages the effective stage delay time is fully $1 + \log n$.

Let $T_{2'}$ denote the computation time for an (n,k) Method 2' network. From what we said above, we have that $T_{2'} = k(1 + \log n) = \hat{T}_2$ if $k \leq \frac{n}{2}$. But since $\hat{c}_n = \frac{n-1}{\log n} < \frac{n}{2}$ (for $n \geq 4$), it follows that the crossover point between Method 1 and Method 2' is simply $\hat{c}_n$ and that for $k < \hat{c}_n$, $T_{2'} = \hat{T}_2 < T_1$. That is, in the case of Method 2' our simple upper bound formulas, time $= k(1 + \log n)$ and crossover $= \frac{n-1}{\log n}$, hold exactly! (Actually, $T_{2'} < k(1 + \log n)$ when $k > \frac{n}{2}$, but of course in that range we would never use Method 2', anyway, since it is slower than Method 1.)

Now recall from our study of Method 2 that c_n, the crossover point between Methods 1 and 2, is slightly greater than $\hat{c}_n$. Remember also that for $k < c_n$, T_2 is slightly less than $\hat{T}_2$. Therefore, over the range of interest Method 2 outperforms Method 2' by a small amount. However, this difference in speed (plus the lower module count of Method 2) must be weighed against the considerable fan-out advantage of Method 2'.

2.6. Summary

As we stated earlier, an (n,k) Method 2 network is faster than the corresponding (n,k) Method 1 network only if k is small compared to n. Note from Table 2 that $\frac{c_n}{n}$ is generally somewhere between .1 and .2. That is, as k varies from 1 to n, only in the lower 10 to 20 percent of that range (depending on n) does Method 2 outperform Method 1. The situation is similar for Method 2', as can be seen from the tabulated values of $\frac{\lfloor \hat{c}_n \rfloor}{n}$.

Figure 8 compares the speeds of Methods 1, 2, and 2' in a general way.

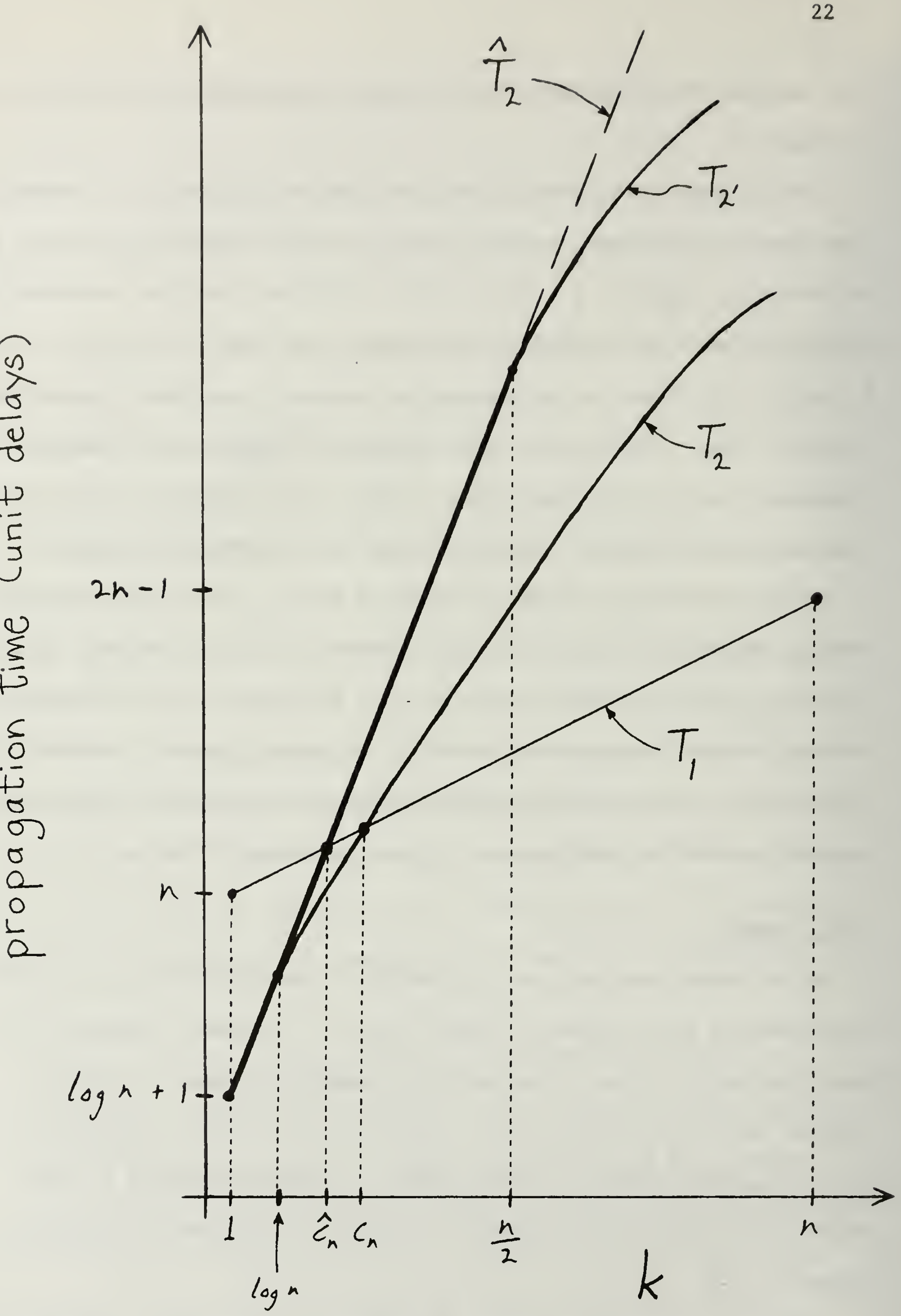

Figure 8. Speed comparison of Methods 1, 2, and 2'.

Considering n to be fixed, we have sketched T_1, $\hat{T}_2$, T_2, and $T_{2'}$, as functions of k, which runs from 1 to n. (For purposes of presentation these curves have been drawn as continuous; of course, in reality k takes on only integral values.) Figures 16 through 19 in Chapter 5 present the same curves plotted from actual data for several values of n. In addition, Table 6 (in Chapter 6) displays side by side the major characteristics of Methods 1, 2, and 2'.

To summarize our results on Method 2 (and its variation, Method 2'), we may say that this technique has definite, though rather limited, advantages. Method 2 is a simple and quite general approach for achieving significant speedup in two-dimensional arrays. However, this speedup is obtained only when the relative dimensions of the array are in a certain narrow range. Thus in spite of the fact that they require more hardware, Method 2 networks are in most cases actually slower than the original (Method 1) rectangular arrays. It is this difficulty that motivated the development of Method 3.

CHAPTER 3
METHOD 3: CONSTRUCTION

Like Method 2, Method 3 transforms an (n,k) rectangular (Method 1) network into a circuit consisting of k stages built one upon the other, each producing a single network output. In both cases the i^{th} stage corresponds to the i^{th} row of the rectangular array. However, in contrast to Method 2, the modules used in Method 3 are identical to those of the original network.

3.1. An Example

To illustrate the new technique, we shall consider an (8,3) Method 1 network; this is shown in Figure 9. Figure 10 depicts the first stage of the corresponding Method 3 network. The modules in both diagrams are the same; however, Figure 10 does not show their * outputs because only the # outputs are used within stage 1. What we have done in Figure 10 is to simply fan in as quickly as possible the inputs a_1 through a_8 plus the input x_1. Output z_1 is thus the #-product of these nine inputs. This is the same as output y_1 in Figure 9, provided that # is associative.

Now note that, except for x_2, each input to the second row of the Method 1 network is a *-product of signals occurring within the first row. Output y_2 is then the #-product of all of these inputs. This suggests that we construct stage 2 of the Method 3 network in the following way. We shall simply take the * outputs from all the stage 1 modules (along with input x_2) and form their #-product as fast as we can. This is shown in Figure 11, where dashed lines indicate the second stage. The * outputs of the stage 2 modules are not shown, since they will not be needed until

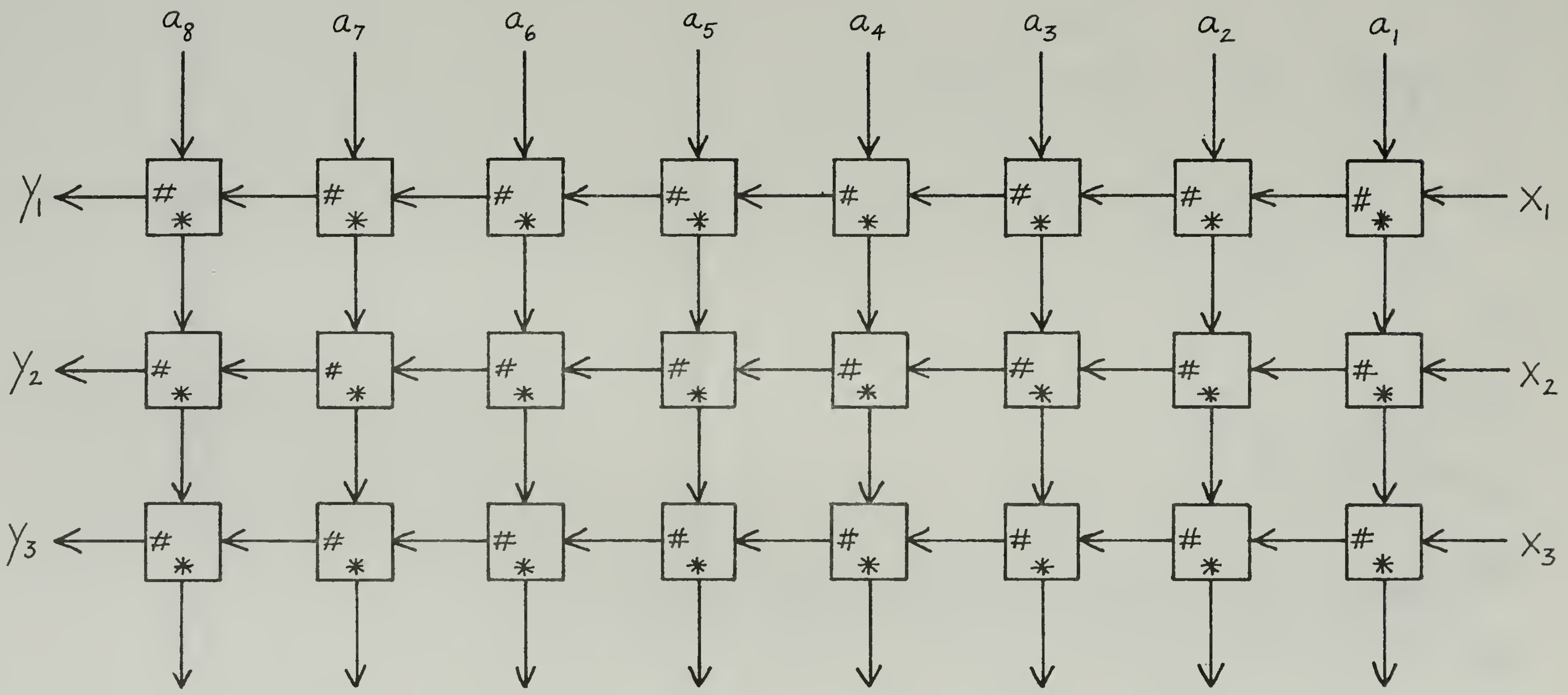

Figure 9. (8,3) Method 1 network.

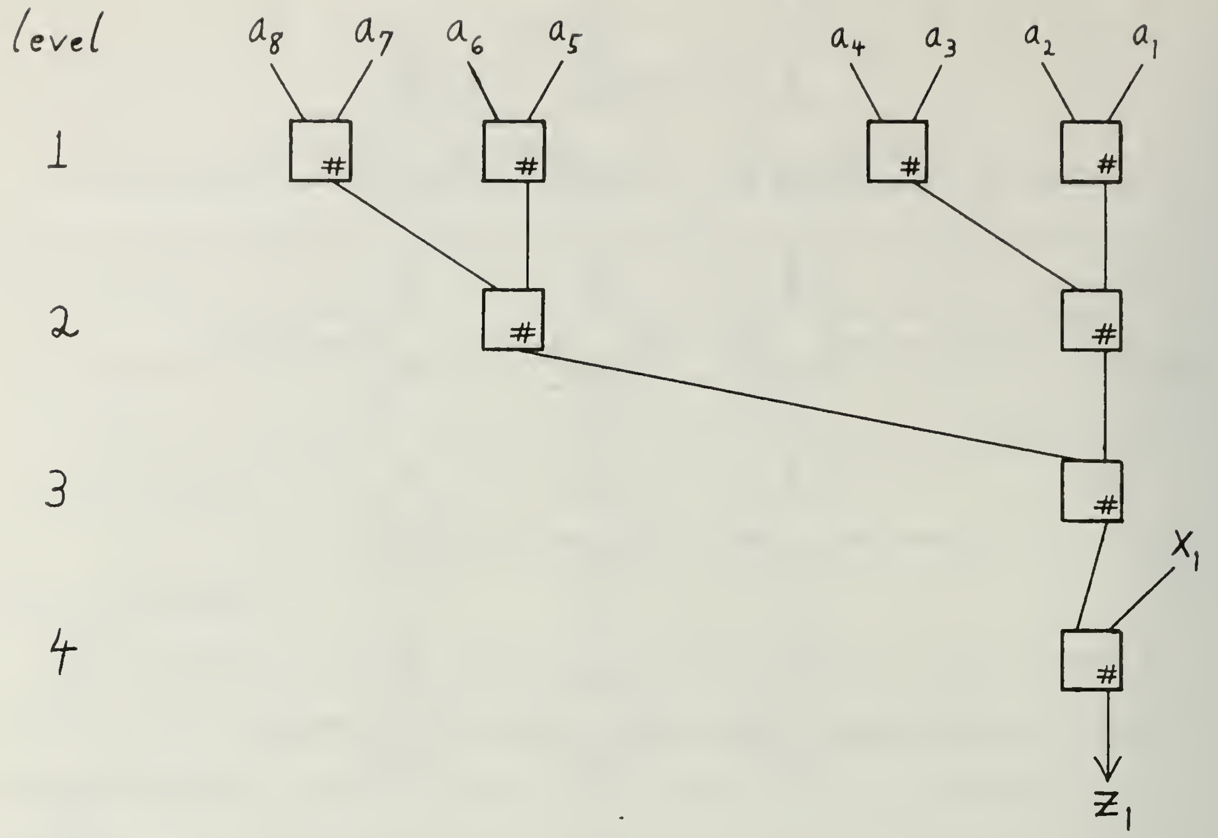

Figure 10. Stage 1 of (8,3) Method 3 network.

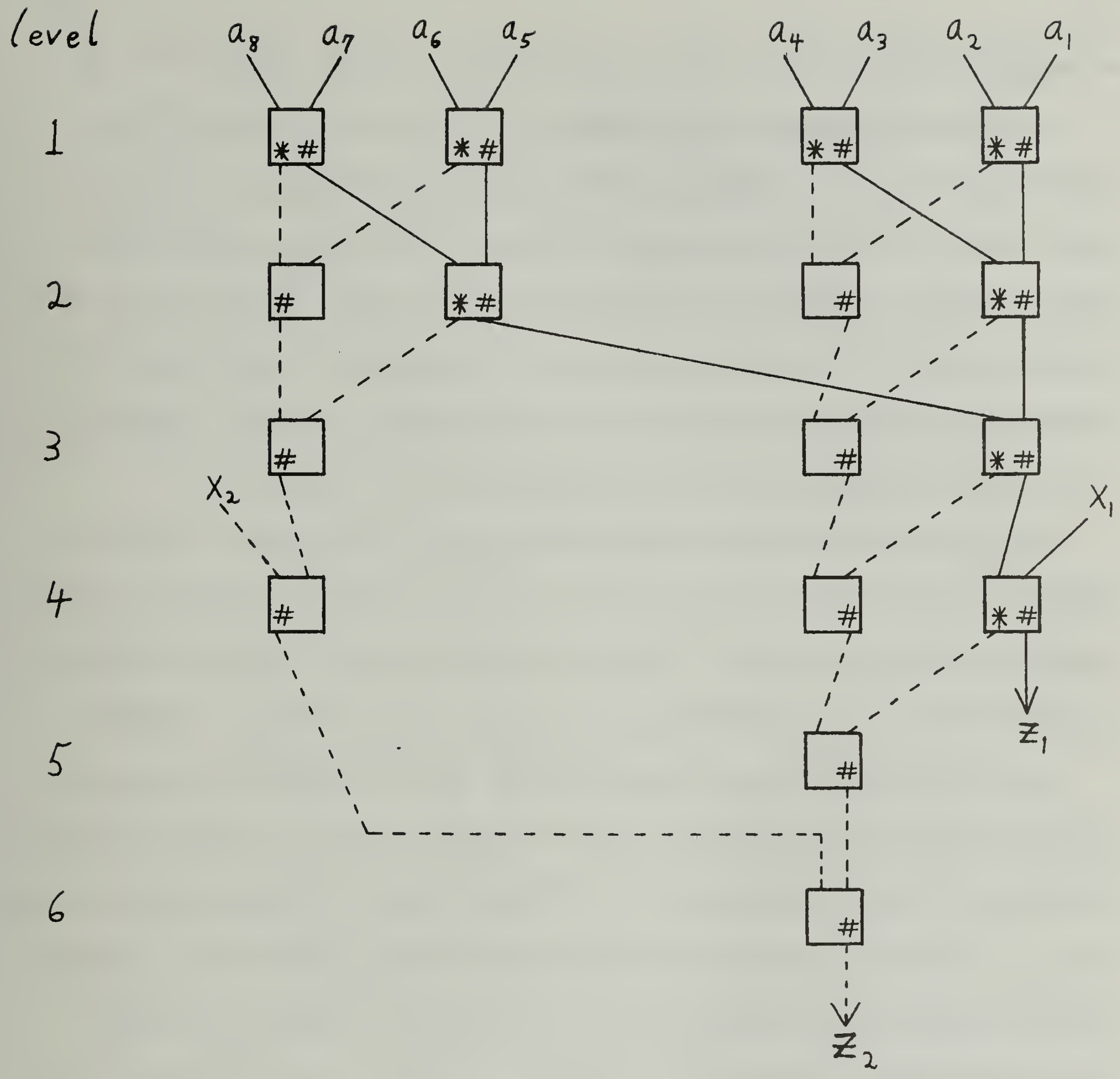

Figure 11. Stages 1 and 2 of (8,3) Method 3 network.

we construct stage 3.

In general the second stage output z_2 will not be the same as the output y_2 in Figure 9. However, as we shall see in Chapter 4, there do exist cases in which not only does z_2 equal y_2, but every stage of the Method 3 network produces the same output as the corresponding row of the Method 1 network. We shall then say that the Method 3 transformation is "output-preserving". The existence of this property, without which Method 3 is useless, depends on how the operations # and * behave.

We need now to introduce some terminology and conventions. Henceforth, we shall use the shorter word "cell" in place of "module". In the Method 3 network, the cells of stage 1 will be called "1-cells", those of stage 2, "2-cells", and in general the cells of stage i will be called "i-cells".

The level numbers listed along the left side of Figure 11 correspond to propagation times within the Method 3 network. That is, the outputs of all cells on level i (regardless of to which stage they belong) become valid exactly i unit delays after the network inputs are applied. For example, the propagation time to output z_1 is 4 unit delays, whereas z_2 becomes valid after 6 unit delays.

Because of the way our figures are drawn, we shall say, for example, that level 3 is "above" level 5. That is, higher level numbers correspond to "lower" levels. The highest level (lowest level number) on which an i-cell occurs will be called the "top" level of stage i. For instance, in Figure 11 level 2 is the top level of stage 2. Similarly, level 4 is the "bottom" level of stage 1. The number of levels containing at least one i-cell will be called the "height" of stage i. For example, the height of stage 1 is 4, whereas stage 2 has height 5.

In constructing stage 2, as well as all other stages, no attention is paid to the order in which we "# together" the * outputs from the previous stage. All that matters is that no time is wasted in performing the computation. As soon as a stage 1 * output becomes available, it is immediately used to compute a valid #-product in stage 2. Similarly, the # outputs of the 2-cells are fanned in to form z_2 as quickly as possible. In this way we obtain a great deal of parallelism, often putting a number of cells from several different stages all on the same level.

Actually, some "wasted" time is unavoidable, for whenever an odd number of signals appears at a given level, one of them, of course, cannot be utilized immediately but must be "reserved" for use at a lower level. For example, in Figure 11 the output of the leftmost cell on level 4 is not used until level 6. In our method the choice of which signal to reserve is made arbitrarily, except that by convention the network inputs x_i are always the first to be reserved. For instance, x_1 and x_2, though available at time zero, are not used until level 4.

All of this will be made more clear when in the next section we present a formal algorithm for constructing Method 3 networks. Meanwhile, let us complete our example by showing in Figure 12 the addition of stage 3, which (like stage 1) is indicated by solid lines. To lessen confusion we have labeled each cell with the number of the stage to which it belongs. The inputs to stage 3 are the * outputs of the 2-cells, together with the network input x_3. The #-product of all of these is formed by the 3-cells and denoted by z_3.

Once again we caution that in general z_3 will not be equal to y_3 in Figure 9. In fact, in most cases the Method 3 outputs will depend on

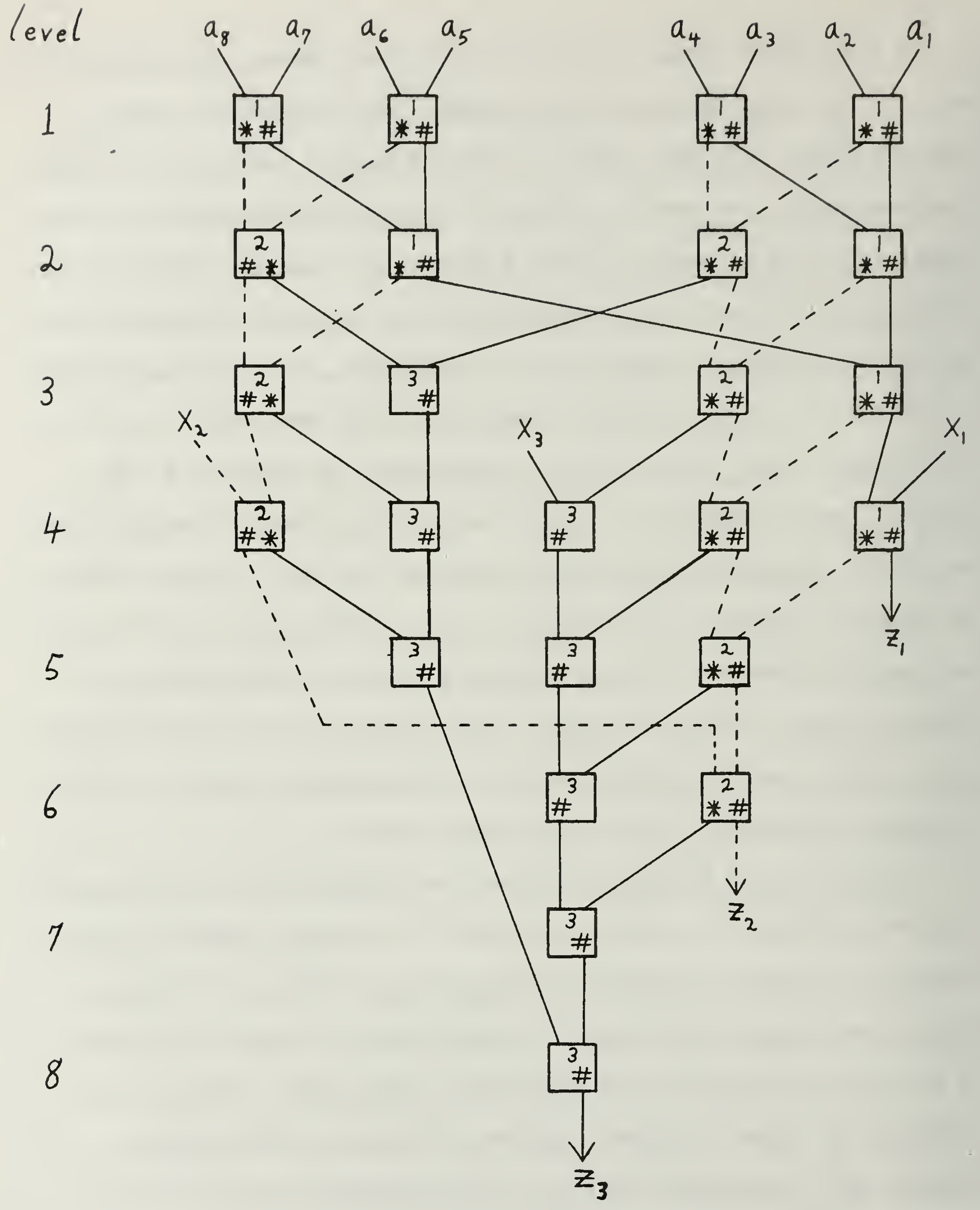

Figure 12. (8,3) Method 3 network.

exactly which way we connected the various cell inputs in constructing each stage. However, our interest lies in those cases for which the outputs are the same regardless of the order in which the products are formed, these outputs being equal to the corresponding Method 1 outputs. This is the "output-preserving" property mentioned earlier.

As examples of our terminology, observe in Figure 12 that stage 3 has top level 3, bottom level 8, and height 6. Also note that the output of the leftmost cell on level 5 is reserved until level 8. However, the most important thing to observe here is that the total computation time for the Method 3 network is 8 unit delays, as opposed to the 10 unit delays Method 1 requires (Method 2 takes 12).

Note that Figure 11 can be viewed as either the first two stages of an (8,3) network or as a complete (8,2) network. Likewise, the array in Figure 12 can be made into an (8,4) Method 3 network by simply adding one extra stage. The * outputs of the 3-cells would be inputs to that stage, which would be constructed in the same way as the previous stages.

As a further illustration of our procedure, Figure 13 depicts a (16,4) Method 3 network. Solid lines are used for stages 1 and 3, dashed lines for stages 2 and 4. The network propagation time is 11 unit delays; this compares to 19 for Method 1 and 20 for Method 2. Chapter 5 is devoted to a detailed discussion of the speedup provided by Method 3.

3.2. The Algorithm

We now present a formal procedure for designing any (n,k) Method 3 array ($1 \leq k \leq n$, n a power of 2). We shall make use of the notation and

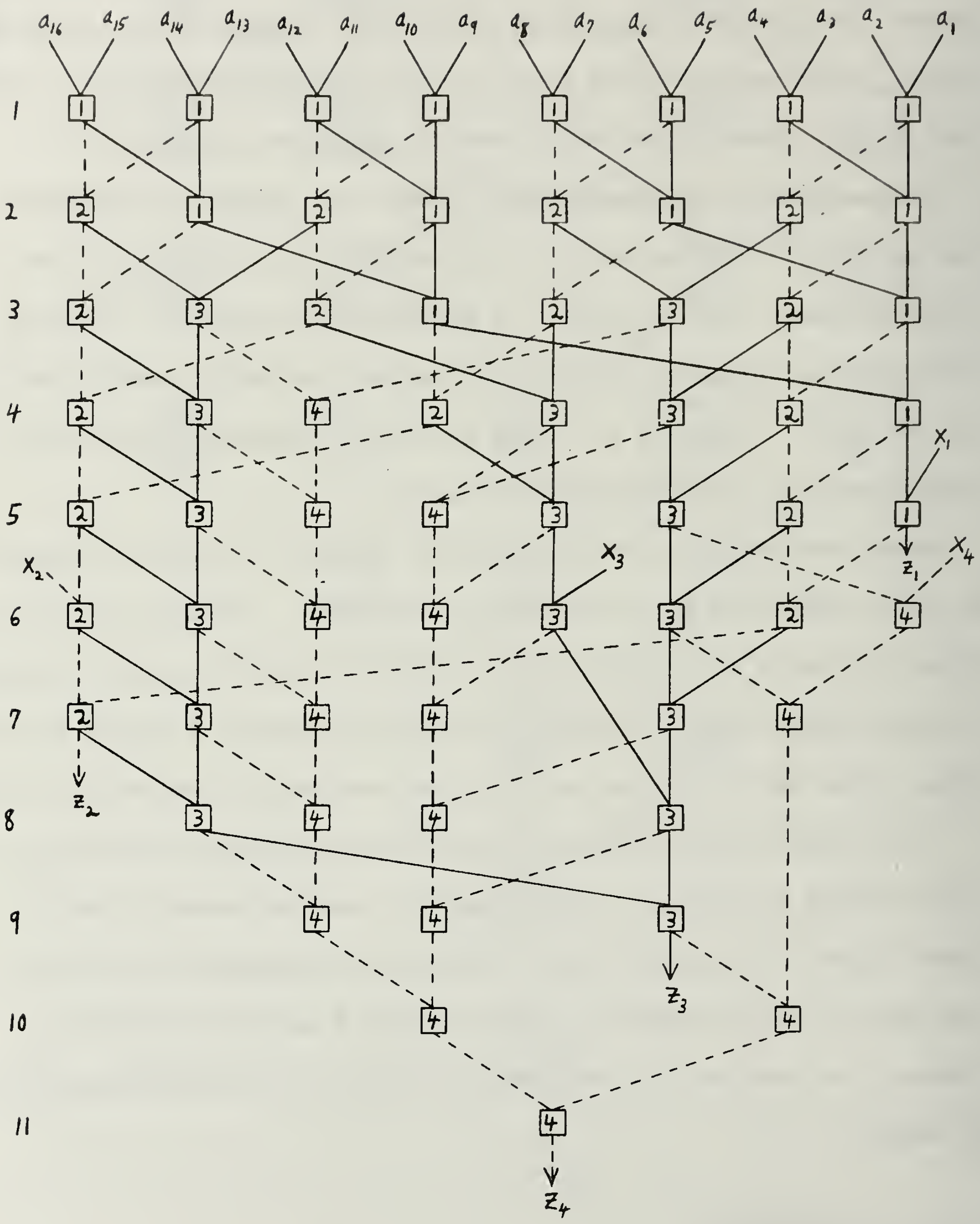

Figure 13. (16,4) Method 3 network.

terminology introduced in describing our example. The algorithm consists of two parts: Step A is performed once, then Step B is iterated k - 1 times (for i from 2 to k). The statement of the algorithm is followed by several remarks which will be useful later in the paper.

<u>Algorithm for Constructing an (n,k) Method 3 Network</u>

STEP A

To construct stage 1:

1. Recalling that n is a power of 2, fan in the inputs $a_1, a_2, \ldots, a_n$ with a full binary tree of cells, using only the # outputs. (The root cell of this tree will be on level log n. See Figure 10 for an example where n = 8.)
2. Install an additional cell on level log n + 1, having as inputs the network input x_1 and the # output from the bottom cell of the above tree. The # output of this added cell is the network output z_1.

STEP B

To construct stage i,
once stage i - 1 has been constructed:

1. "Reserve" the input x_i.
2. Let L be the number of the top level of stage i - 1.
3. Let S be the set of all * outputs from (i - 1)-cells on level L and all # outputs from i-cells on level L. Let C denote the cardinality of S.

4. If C is odd, then do the following:

   If there is a reserved element, add it to S; otherwise, remove an element from S and reserve it.

   C is now even or zero.

5. If C is not zero, then do the following:

   a. Install $\frac{C}{2}$ i-cells on level L + 1, connecting their inputs to the elements of S (in any order whatever).

   b. Increment L by one.

   c. Go to step 3.

6. Now C = 0 and we are done. The reserved element is the network output z_i.

- - - - - - - - - - - - - -

REMARKS (valid for all i)

<u>Remark 1</u> The top level of stage i is level i. (This is obviously true for i = 1, and an easy induction verifies it for i > 1.)

<u>Remark 2</u> Note that stage i skips no levels; that is, it has at least one cell on every level between its top level and its bottom level.

<u>Remark 3</u> Clearly, the bottom level of stage i is below the bottom level of stage i - 1 and contains exactly one i-cell (whose # output is z_i).

<u>Remark 4</u> Hence, the total number of levels in the network is equal to the bottom level number of stage k. This is the propagation time through the network (in unit delays).

Remark 5 Since each i-cell has two input lines and one # output line, stage i forms a regular binary tree. The branches of this tree are the input lines and # output lines of all the i-cells, its interior nodes are the i-cells themselves, and (if $i > 1$) its leaves are the $(i - 1)$-cells together with the input x_i.

3.3. Required Hardware

It has been mentioned, of course, that the cells of Method 3 are identical to those used by Method 1. We now proceed to show that corresponding Method 1 and Method 3 networks contain exactly the same number of cells, as well.

Theorem 1

Each stage of an (n,k) Method 3 network contains exactly n cells.

Proof

By induction. We first show the theorem is true for stage 1. In step A1 of the Method 3 algorithm, we construct a full binary tree fanning in n inputs, where n is a power of 2. This, of course, requires $n - 1$ cells. In Step A2 we complete stage 1 by installing one additional cell. Thus stage 1 has exactly n cells.

Now suppose the theorem is true for stage $i - 1$. Let us consider stage i. By Remark 5, the i-cells are the interior nodes of a regular binary tree whose leaves are the $(i - 1)$-cells plus input x_i. Since by hypothesis there are n $(i - 1)$-cells, this tree has a total of $n + 1$ leaves. Thus, by a well-known result of graph theory, the tree must have n interior nodes. That is, there are exactly n i-cells, whence the theorem is true for stage i.

QED

Since an (n,k) Method 3 network has k stages, by Theorem 1 the total number of cells it contains is nk, precisely the same as for Method 1. Thus, in contrast to Method 2, Method 3 requires no more hardware than the original rectangular array. Moreover, it is evident from the algorithm that in a Method 3 network each cell output is connected to (at most) one cell input. This is also the case in a Method 1 array. Thus neither method has a fan-out problem, which again is different from Method 2.

We therefore conclude that the Method 3 transformation of a rectangular network amounts to nothing more than a rather limited rearrangement of the interconnections among the nk cells. That this usually results in a significant increase in speed (see Chapter 5) is indeed remarkable.

CHAPTER 4
METHOD 3: APPLICATIONS

Having just boasted the merits of Method 3, we must again remind the reader that unfortunately it cannot always be used, for often the transformation is not output-preserving. However, in this chapter we shall present several cases for which Method 3 does indeed work. These are intended as illustrative examples only; our array methods are not necessarily the most practical or efficient circuits available to perform the given computations. We shall also look at some cases where Method 3 does not work and then speculate as to which properties of the cell operations are relevant to output-preservation.

4.1. The Augmented Parallel Counter

In this example the interconnections of the Method 1 array (Figure 1) are single bit lines, and the cell operations (Figure 2) are as follows: # is EXCLUSIVE OR (XOR) and * is logical AND. Each cell is thus a half adder (see Figure 14).

To understand what a rectangular network of half adders does, let us look, for instance, at Figure 9. Suppose first that $x_1 = x_2 = x_3 = 0$. Then y_1 is just the EXCLUSIVE OR sum of the inputs $a_1, a_2, \ldots, a_8$. The carries from this summation are then XOR'ed together to form y_2, and finally y_3 is the EXCLUSIVE OR of the carries from the y_2 summation. Thus $y_3y_2y_1$ is simply the ordinary sum, in weighted binary form, of the eight inputs a_i. In other words, the three-place binary numeral $y_3y_2y_1$ represents the total number of a_i inputs which are in the logic ONE state.

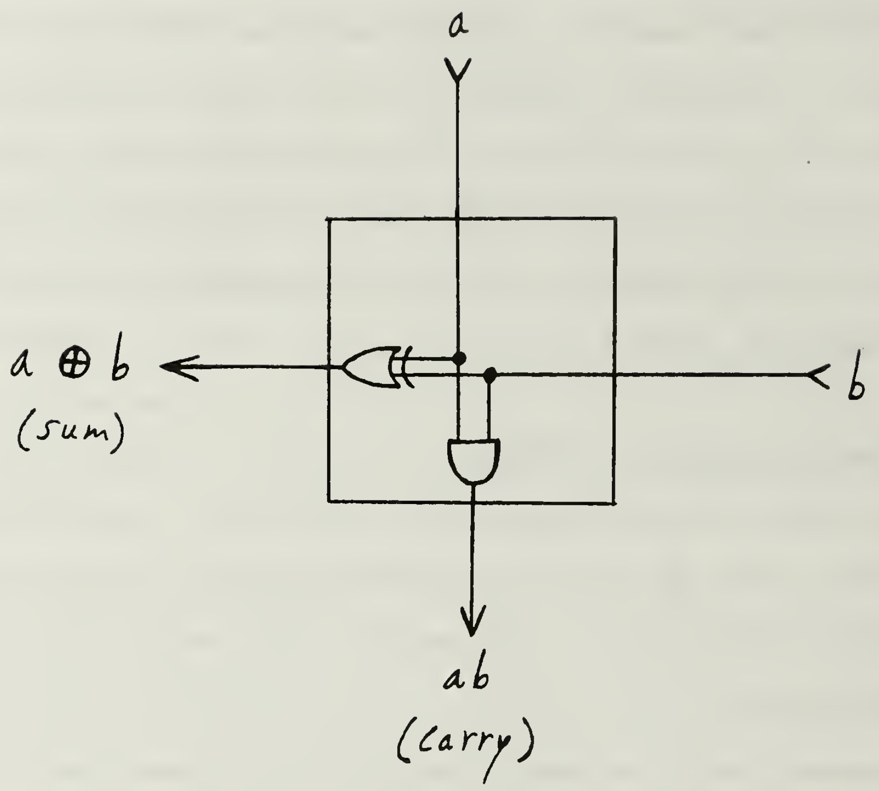

Figure 14. Half adder.

Hence, the network acts as an eight-input parallel counter.

However, the three-bit output can represent only 8 of the 9 possible a_i sums. That is, $y_3y_2y_1$ is actually the sum of the a_i's modulo 8. Moreover, dropping our assumption that the x_i's are zero, we see that the array output in fact represents the modulo 8 sum of the weighted binary number $x_3x_2x_1$ and the number of active (logic ONE) a_i inputs.

Thus an (n,k) Method 1 network of half adders is a kind of generalized parallel counter. Its k-bit output represents, in binary coded form and modulo 2^k, the total number of the n top inputs which are active, plus the weighted binary number formed by the k right-hand inputs. We shall call such a circuit an "augmented parallel counter". If $k = \log n + 1$ and the x_i are all zero, then the array becomes an ordinary n-input parallel counter -- it simply computes the number of a_i's which are ONE.

Now to see what happens when the Method 3 transformation is applied to an augmented parallel counter, let us return to our previous example, the (8,3) network (Figure 9). Recalling that # is XOR, we see from Figure 10 that z_1, the first Method 3 output, is the XOR of x_1 and all the a_i's. That is, z_1 is the least significant bit of the sum of the eight a_i's and the binary number $x_3x_2x_1$. Since * is AND, Figure 11 shows that z_2 is the XOR of x_2 and all the carries from the z_1 summation. Hence, z_2 is the next-to-least significant bit of the sum of $x_3x_2x_1$ and all the a_i's. Similarly, z_3 is the next most significant bit of this sum (see Figure 12). Thus we see that $z_3z_2z_1 = y_3y_2y_1$; that is, the Method 3 network performs exactly the same computation as does the original Method 1 array.

Clearly, this argument can be generalized to any (n,k) augmented parallel counter. The Method 1 network and the Method 3 network will always compute the same sum, column by column. The two methods are dissimilar only in that for any one output bit they produce the sum in a different order, the Method 3 arrangement being faster. We therefore conclude that when # is XOR and * is AND, the Method 3 transformation is output-preserving.

4.2. The Partial Sorter

Unlike the previous example, in which the cell operations were Boolean functions, many of the other cases we shall look at feature cell functions which operate on integer values. We will then assume that in our arrays each interconnecting line is capable of carrying a representation of any positive integer. Naturally, there would in practice be a limit to the size of the integers, but this restriction has no bearing on the relevance of our examples.

For our next special case, we take the cell operation # to be MAX and the operation * to be MIN. Of course, MAX returns the larger of its two integer operands, while MIN returns the smaller.

To see what a rectangular array of such cells does, let us look once again at Figure 9. The network inputs are now positive integers. Clearly, y_1 will be the largest of the 9 numbers x_1, a_1, a_2, . . . , a_8. The other 8 numbers are passed down to the second row of the array. They are then compared with x_2, and the largest of <u>these</u> 9 integers becomes y_2. A similar operation determines y_3. Hence, we see that the network outputs will simply be the 3 largest numbers among the 11 numbers which were inputs to the array. It is also easy to see that if the x_i's were originally in order, with x_1 largest and x_3 smallest, then the y_i's will likewise be in order.

Of course, similar reasoning holds for any (n,k) Method 1 network composed of such cells. The output numbers are always the k largest of the n + k input numbers. Moreover, if the right-hand inputs are in increasing order from the bottom to the top of the array, then the same will be true of the outputs. Thus the network is able to take an ordered list of length k and an unordered list of length n and merge them into an ordered list of length k composed of the largest elements in the original two lists. Hence, it is with some justification that we call such a network a "partial sorter".

It is quite easy to convince oneself that in this case, too, Method 3 is output-preserving. Look, for example, at Figure 12 and recall that for each i the # (i.e., MAX) outputs of the i-cells are kept within stage i to eventually produce output z_i, while the * (i.e., MIN) outputs are passed on to stage i + 1. Hence, the Method 3 network operates in exactly the same way as does the Method 1 network, producing the largest k numbers at its outputs. Only the order of comparison is different.

4.3. Other Special Cases

Having presented two examples in detail, we shall now simply give a list of a number of other particular cases that were investigated. They employ as cell operations a wide variety of two-variable functions, some Boolean and some integral. Several of these -- XOR, AND, MAX, and MIN -- we have used before; another one of them is just the usual logical OR. The rest require a brief explanation.

There are three new Boolean functions. COIN, short for "coincidence", returns a 1 if and only if its two arguments are equal. IMP, which stands for "implies," has value 1 for all cases except 1 IMP 0. Finally, FIR, short for "first," simply returns its first argument; i.e., x FIR y = x.

We shall also use a number of operations defined on the set of positive integers. The value of GCD is the greatest common divisor of its two operands. Similarly, LCM is the least common multiple. The function ADD is just ordinary addition, while MULT performs ordinary multiplication. Lastly, AVG returns the average of its two arguments; strictly speaking, this must be defined on the set of rationals, since the average of two integers is not always an integer.

We combined these functions into various #/* pairs and determined for which of these Method 3 is output-preserving. The results of this effort are shown in Table 3. For the moment ignore the column marked "conjecture properties violated"; this will be explained in the next section.

Two of the sixteen cases have already been described: example no. 1 is the augmented parallel counter and example no. 4 is the partial sorter. Note that we list four other cases for which Method 3 works. For these, the proofs of output-preservation will be omitted. As for those instances where Method 3 fails, we merely found simple counterexamples by plugging in specific inputs and computing the outputs by hand. In fact, in all cases (4,3) networks sufficed to demonstrate the lack of output-preservation.

4.4. A Conjecture

We say that the cell operation * "distributes over" the cell operation #

B = Boolean
I = (positive) integer
R = (positive) rational

example no.	cell operations #	cell operations *	operand type	output-preserving?	conjecture properties violated
1	XOR	AND	B	yes	none
2	AND	OR	B	yes	none
3	COIN	OR	B	yes	none
4	MAX	MIN	I	yes	none
5	GCD	LCM	I	yes	none
6	GCD	GCD	I	yes	none
7	AND	XOR	B	no	not distributive, XOR not idempotent
8	COIN	AND	B	no	not distributive
9	IMP	OR	B	no	IMP not associative, not commutative
10	OR	FIR	B	no	FIR not commutative
11	ADD	MULT	I	no	MULT not idempotent
12	ADD	MIN	I	no	not distributive
13	MAX	ADD	I	no	ADD not idempotent
14	ADD	ADD	I	no	not distributive, ADD not idempotent
15	MULT	GCD	I	no	not distributive
16	MAX	AVG	R	no	AVG not associative

Table 3. Attempts to apply Method 3.

if a * (b # c) = (a * b) # (a * c) for all a, b, c in the set S of operands. (If * is not commutative, we also require that (b # c) * a = (b * a) #(c * a).) To make a less familiar definition, we shall say that the operation * is "idempotent" if a * a = a for all a in S.

A large part of our research has been an attempt to characterize those rectangular arrays to which Method 3 can be successfully applied. That is, we have tried to find sets of conditions necessary and/or sufficient for the Method 3 transformation to be output-preserving. Experience with many examples as well as consideration of several theoretical aspects have led us to propose the following:

CONJECTURE

Method 3 is output-preserving if and only if the cell operations # and * have the following properties:

a. # is associative and commutative

b. * is associative, commutative, and idempotent

c. * distributes over #

Although this statement has stubbornly resisted all attempts to prove it, no counterexamples have yet been found. The rightmost column in Table 3 shows for each case those properties listed in the conjecture which are not satisfied by the given pair of cell operations. Any property not mentioned was proved to hold. The examples marked with "none" are those which satisfy all the conditions of the conjecture. Note that these are precisely the cases for which Method 3 is output-preserving. The examples where Method 3 fails are those which violate one or more of the conjecture properties. Hence, Table 3 is consistent with our belief that these properties are both necessary and sufficient.

Since the Method 3 algorithm pays no attention to the order in which inputs are combined, it is not surprising that in order for the outputs to be the same as in Method 1, both cell operations must be associative and commutative. (In fact, this has been proved in the case of the # operation.) However, the need for distributivity and, especially, for idempotence is much less obvious.

Note that Table 3 concerns arrays of arbitrary dimensions in which the cell operations are specified. While trying to prove the necessity of the conjecture conditions, we investigated networks with arbitrary cell functions but where n and k were small enough to conveniently handle. In manual computations with these networks, we needed to utilize all of the conjecture properties in order to get the outputs of Methods 1 and 3 to be equal. Nevertheless, to actually prove necessity seemed impossible. Perhaps one can somehow construct rather artificial # and * operations which violate the conjecture conditions but for which Method 3 works. However, judging from our research it appears doubtful that such a case would arise naturally, that is to say, when using any standard sort of functions for cell operations.

The situation is similar with regard to sufficiency. When working with networks of feasibly small dimensions, we found the conjecture properties sufficient to obtain output equality. A general proof, though, has remained elusive. However, as an indication that the conjecture conditions could well be sufficient, let us point out that they form a more powerful set of postulates than may be evident at first glance. For example, using only those properties we can derive the following formula (for all a, b in S):

$$
\begin{aligned}
(a \# b) * (a \# b) &= [(a \# b) * a] \# [(a \# b) * b] \\
a \# b &= [a * (a \# b)] \# [b * (a \# b)] \\
&= (a * a) \# (a * b) \# (b * a) \# (b * b) \\
&= a \# (a * b) \# (a * b) \# b \\
a \# b &= a \# b \# (a * b) \# (a * b)
\end{aligned}
$$

Moreover, letting a = b and once again using idempotence, we obtain:

$$a \# a = a \# a \# a \# a \qquad (1)$$

Now, it may perhaps be possible to invent a pair of functions satisfying the conjecture conditions but for which Method 3 is not output-preserving; however, once again we consider it very unlikely that one would run into such a counterexample by accident. For one thing, any system which has all the conjecture properties must also satisfy (1) above. Hence, quite probably it possesses as well one of these two common properties (for all a in S):

<u>Property 1</u> $a \# a = 0$, where 0 is an identity element for the # operation

<u>Property 2</u> $a \# a = a$ (i.e., # is idempotent)

Indeed, it is easy to check that one of these holds for every example in Table 3 which satisfies all the conjecture conditions. (Obviously, both properties cannot hold at once, unless 0 is the only element in S.)

Now the point is that either of these two properties, when added to the conjecture conditions, makes the two-operation system even more rich in structure, allowing equations with # and * to be greatly simplified. This seems to enhance the likelihood that corresponding Method 1 and Method 3 outputs are indeed equal.

Taking all of this into account, we come to the following conclusions. In the first place, our conjecture is very plausible. Moreover, even if it is not literally true, the conjecture appears at any rate to be a highly reliable test for determining to which arrays Method 3 can be applied. Because the conjecture properties are easy to check, this provides a quite practical method for screening prospective function pairs. Of course, to be completely safe, one must supply for each individual case an independent proof or disproof of the output-preservation property. Thus we hope that future research will lead to a proof of the conjecture, a proof of some modification of it, or at least some enlightening counterexamples.

4.5. Additional Remarks

Note that all of the conjecture properties hold in any Boolean ring as well as in any Boolean algebra. (For definitions and an excellent discussion of these two types of systems, see the book by Burton.[14]) Thus we would expect Method 3 to be output-preserving for both kinds of systems. This is illustrated by the first two examples in Table 3: XOR/AND is a Boolean ring, while AND/OR is a Boolean algebra. On the other hand, MAX/MIN, for which Method 3 also works, does not belong to either class of systems. Finally, it is interesting to note that

Boolean rings satisfy Property 1, whereas Boolean algebras satisfy Property 2.

If the conjecture is true, then the range of applications for Method 3 may be rather limited, for it seems that the conjecture conditions are really quite restrictive. For example, among the 16 two-variable switching functions, only AND and OR are both commutative and idempotent. Moreover, on the domain of positive integers, the only operations we could readily find satisfying all three conditions on the * function were MAX, MIN, LCM, and GCD. Note that these four functions are essentially the same! (For instance, to find the least common multiple of two numbers, we simply choose the maximum exponent for each prime factor; GCD and MIN are likewise related.)

Clearly, idempotence is the most difficult condition to meet. The other conjecture properties hold in any commutative ring, a very common type of system. Hence, to find applications for Method 3 we might be tempted to search among ordinary commutative rings, hoping that the conjecture's requirement for idempotence is not absolute. Unfortunately, the following proposition has been proved (we omit the argument, which is very tedious): If the cell operations # and * form a commutative ring (with identity), where # is addition and * is multiplication, and if for this system Method 3 is output-preserving, then the operation * is idempotent, that is, the system is actually a Boolean ring.

This result is quite significant in view of the fact that there are not, so to speak, many Boolean rings around. Indeed, according to the famous Stone Representation Theorem, every Boolean ring is isomorphic to a ring of sets! (In a ring of sets, addition is the symmetric difference operation and multiplication is set intersection. See Burton[14] for details.) On the other hand, in order for Method 3 to work it is not necessary that the #/* system be a ring at all; MAX/MIN is a case in point. Hence, the true extent to which Method 3 can be applied is still an open question.

CHAPTER 5

METHOD 3: COMPUTATION TIME

We come now to the crucial question: just how fast does Method 3 perform? Let T_3 be the propagation time in unit delays for an (n,k) Method 3 network. Usually we shall consider n fixed and observe how T_3 varies as k runs from 1 to n. Unfortunately, an exact formula for T_3 in terms of n and k remains to be discovered. However, a combination of theoretical and empirical research has revealed a great deal about the behavior of the function T_3, including simple and accurate approximations valid at least for the range $n \leq 1024$. Moreover, we shall show that T_3 is in general much smaller than both T_1 and T_2, the propagation delays for Methods 1 and 2, respectively.

5.1. Comparison with Method 1

Consider an (n,k) Method 3 network and let h_i denote the height of stage i (as defined in Section 3.1). Note that by Step A of the Method 3 algorithm (Section 3.2), $h_1 = \log n + 1$.

Now consider stage k, the final stage in the network. By Remark 1 (Section 3.2), there are k - 1 levels above stage k. Hence, by Remark 4, the total number of levels in the network is $k - 1 + h_k$. We thus have:

Remark 6 $\quad T_3 = h_k + k - 1$

Hence, surprisingly, the computation time of the Method 3 array depends solely on the height of its last stage. Unfortunately, it turns out that h_k is in general difficult to determine. Much can be gained, however, by studying the properties of the sequence $h_1, h_2, \ldots, h_k$. In fact, we shall now present a few lemmas on the subject, which lead ultimately to a theorem comparing the speeds of Methods 1 and 3.

<u>Lemma 1</u> $h_i \leq n$ (for all i, $1 \leq i \leq k$)

Proof

By Theorem 1 (Section 3.3), stage i contains n cells. Hence, noting Remark 2 (Section 3.2), it is obvious that the height of stage i can be no more than n.

QED

<u>Lemma 2</u> $h_i \leq h_{i+1}$ $(1 \leq i \leq k - 1)$

Proof

By Remark 1, stage i + 1 begins one level below the top level of stage i; by Remark 3, it ends at least one level below the bottom level of stage i. Thus the height of stage i + 1 must be at least as great as the height of stage i.

QED

<u>Lemma 3</u> $h_i = h_{i+1} \Longleftrightarrow h_i = n$ $(1 \leq i \leq k - 1)$

Proof

$\Longleftarrow$

If $h_i = n$, then by Lemma 2 $h_{i+1} \geq n$. On the other hand, by Lemma 1 $h_{i+1} \leq n$. Hence, $h_{i+1} = n = h_i$.

$\Longrightarrow$

Suppose $h_i = h_{i+1}$. Then since (by Remark 1) the top level of stage i + 1 is one level below the top level of stage i, it follows that the bottom level of stage i + 1 is one level below the bottom level of stage i. Let L be the bottom level of stage i + 1. By Remark 3, level L contains exactly one (i + 1)-cell; call it C_L.

Since level L - 1 is the bottom level of stage i, it contains an i-cell, say c_{L-1}; moreover, (by Remark 2) it contains at least one (i + 1)-cell, say C_{L-1}. Now according to the Method 3 algorithm, the * output of c_{L-1} and the # output of C_{L-1} must each be connected to some (i + 1)-cell below level L - 1. But the only (i + 1)-cells below level L - 1 is C_L. Hence, one input of C_L is attached to c_{L-1} and the other to C_{L-1}. This means that there can be no (i + 1)-cell besides C_{L-1} on level L - 1, for there are no available (i + 1)-cells below that level to accept their # outputs.

Now consider level L - 2. It contains at least one i-cell c_{L-2} and at least one (i + 1)-cell C_{L-2}. Since C_{L-1} is the only available (i + 1)-cell below level L - 2, its two inputs must be connected to c_{L-2} and C_{L-2}. Hence, there can be no (i + 1)-cells besides C_{L-2} on level L - 2.

Clearly, this argument can be repeated for levels L - 3, L - 4, etc., all the way to the top level of stage i + 1. The result is that every level of stage i + 1 has only one (i + 1)-cell. Since (by Theorem 1) the stage has n cells, its height must be n; that is, $h_{i+1} = n$. But then since $h_i = h_{i+1}$, we get $h_i = n$.

QED

Recalling that $h_1 = \log n + 1$ and taking into account Lemmas 1 through 3, we see that the sequence $\{h_i\}$ starts at $\log n + 1$, strictly increases until it reaches n, and remains at n thereafter. Let s_n be the smallest value of i for which $h_i = n$. We can then list the basic properties of the sequence $\{h_i\}$ as follows:

Property A	$h_1 = \log n + 1$
Property B	$\{h_i\}$ strictly increases for $1 \leq i \leq s_n$
Property C	$h_i = n$ for $i \geq s_n$

Note that the height h_i of a stage is a kind of inverse measure of its parallelism. Since (by Theorem 1) each stage has exactly n cells, the greater the height of a stage, the less parallel is its computation. Stage 1, with height $\log n + 1$, is as parallel as it can be. By Property B, later stages gradually become less parallel. Beyond some point (s_n by Property C), each stage has height n, i.e., its computation is completely serial.

We have now assembled enough information to easily prove our theorem. Recall that for an (n,k) Method 1 network, the computation time $T_1 = n + k - 1$. T_3 is the computation time of the corresponding Method 3 network.

Theorem 2

a. for $k < s_n$, $T_3 < T_1$

b. for $k \geq s_n$, $T_3 = T_1$

Proof

By Remark 6, $T_3 = h_k + k - 1$. By Property C, for $k \geq s_n$, $h_k = n$, whence $T_3 = n + k - 1 = T_1$, which proves b. By Properties B and C, for $k < s_n$, $h_k < n$, whence $T_3 < n + k - 1 = T_1$, proving a.

QED

Therefore, in all cases Method 3 is at least as fast as Method 1. Moreover, s_n is a sort of "crossover" point between Methods 1 and 3, analogous to the crossover point c_n between Methods 1 and 2. For $k < s_n$, Method 3 outperforms Method 1. Note that since $T_1 - T_3 = (n + k - 1) - (h_k + k - 1) = n - h_k$, it follows from Property B that this speed difference strictly decreases as k runs from 1 to s_n. Beyond s_n, the two methods have identical speed.

Recalling how s_n was defined, we can rephrase our result as follows: If every stage of the Method 3 network has some amount of parallelism, then the network outperforms the Method 1 array; but if one or more of the later stages is serial, then the two methods have equal performance.

Of course, Theorem 2 leaves us with the question of what the value of s_n is (for each n). We would also like to know more about how T_3 behaves when $k < s_n$. These matters will be addressed in the following sections.

5.2. Further Theoretical Results

We shall shortly present theorems which provide more details about the speed of Method 3. First, however, we must prove two additional lemmas concerning the sequence $\{h_i\}$ of stage heights.

Lemma 4

for $1 \le i \le s_n$, $\log n + i \le h_i \le n - s_n + i$

Proof

By Property A, $h_1 = \log n + 1$. Thus by Property B we must have:

$$h_2 \ge \log n + 2$$

$$h_3 \ge \log n + 3$$

$$\vdots$$

$$h_{s_n} \ge \log n + s_n$$

This proves the first inequality.

By Property C, $h_{s_n} = n$. Thus by Property B we must have:

$$h_{s_n-1} \le n - 1$$

$$h_{s_n-2} \le n - 2$$

$$\vdots$$

$$h_1 = h_{s_n-(s_n-1)} \le n - (s_n-1)$$

Hence, $h_{s_n - j} \leq n - j$ for $0 \leq j \leq s_n - 1$. To change subscripts, let $i = s_n - j$. Then $1 \leq i \leq s_n$, and our result becomes $h_i \leq n - (s_n - i) = n - s_n + i$. This proves the second inequality.

QED

Lemma 5

Suppose that for some $I \leq s_n$, $h_I = n - s_n + I$. Then for all i where $I \leq i \leq s_n$, $h_i = n - s_n + i$.

Proof

If $h_I = n - s_n + I$ for some $I \leq s_n$, then by Property B we must have:

$$\begin{aligned} h_{I+1} &\geq n - s_n + I + 1 \\ h_{I+2} &\geq n - s_n + I + 2 \\ &\vdots \\ h_{s_n} &\geq n - s_n + s_n \end{aligned}$$

That is, for all i where $I \leq i \leq s_n$, $h_i \geq n - s_n + i$. But by Lemma 4 we also have $h_i \leq n - s_n + i$.

QED

From Lemma 4, we note in particular that:

$$\begin{aligned} \log n + i &\leq n - s_n + i \\ \log n &\leq n - s_n \\ s_n &\leq n - \log n \end{aligned}$$

If s_n were equal or very close to this upper bound, then since n - log n is in general close to n, s_n would also be close to n. By Theorem 2 (Section 5.1), this would mean that Method 3 outperforms Method 1 for most k between 1 and n.

Thus in some sense n - log n is an ideal value for s_n. For that reason we introduce the following special notation:

$$\hat{s}_n \equiv n - \log n$$

$$d_n \equiv \hat{s}_n - s_n$$

Our hope is that d_n is small (relative to n).

In preparation for the upcoming theorems, we must make another definition:

$$\hat{T}_3 \equiv \log n + 2k - 1$$

Moreover, we note the following equality:

Remark 7 $\qquad \hat{T}_3 + d_n = n - s_n + 2k - 1$

This is a direct consequence of the above definitions and will be used often in the work that follows.

Theorem 3

$$\text{for } 1 \leq k \leq s_n, \quad \hat{T}_3 \leq T_3 \leq \hat{T}_3 + d_n$$

Proof

From Lemma 4 we have:

$$\log n + k \leq h_k \leq n - s_n + k$$

Adding k - 1, we get:

$$\log n + 2k - 1 \leq h_k + k - 1 \leq n - s_n + 2k - 1$$

But then using Remarks 6 and 7, we obtain:

$$\hat{T}_3 \leq T_3 \leq \hat{T}_3 + d_n$$

QED

This theorem provides another reason for hoping that $d_n = \hat{s}_n - s_n$ is small; for if it is, then the simple function $\hat{T}_3 = \log n + 2k - 1$ is a good approximation to T_3 over the range $1 \leq k \leq s_n$. Of course, when $k > s_n$ we know by Theorem 2b that $T_3 = T_1 = n + k - 1$. Moreover, d_n small means that $\hat{s}_n = n - \log n$ is an accurate estimate for s_n. Hence, d_n is a crucial parameter. If it is small compared to n, then to a good approximation we know the values of T_3 for all k, plus we know (as mentioned earlier) that $T_3 < T_1$ for all k except those very close to n. We postpone to the next section our evidence that d_n is indeed quite close to zero.

We can improve our results even more by introducing another parameter. Note that when $k = s_n$, we have:

$$\begin{aligned} T_3 &= T_1 && \text{(by Theorem 2b)} \\ &= n + k - 1 \\ &= n - k + 2k - 1 \\ &= n - s_n + 2k - 1 \\ &= \hat{T}_3 + d_n && \text{(by Remark 7)} \end{aligned}$$

Thus we can define e_n to be the smallest value of k for which $T_3 = \hat{T}_3 + d_n$. Obviously, $e_n \le s_n$.

Theorem 4

For $e_n \le k \le s_n$, $T_3 = \hat{T}_3 + d_n$

Proof

Using Remarks 6 and 7, we find that:

$$T_3 = \hat{T}_3 + d_n \iff h_k + k - 1 = n - s_n + 2k - 1$$

$$\iff h_k = n - s_n + k$$

Hence, by the definition of e_n, $h_{e_n} = n - s_n + e_n$. But then according to Lemma 5, when $e_n \le k \le s_n$, $h_k = n - s_n + k$, whence (from the above) $T_3 = \hat{T}_3 + d_n$.

QED

Thus for values of k between e_n and s_n, T_3 is given exactly by the simple expression $\log n + 2k - 1 + d_n$. Since for $k > s_n$, $T_3 = n + k - 1$, only for $k < e_n$ is a formula for T_3 not known (and even in that case the bounds of Theorem 3 apply). Hence, it would be nice if e_n were relatively small. In the next section we shall find that this is apparently the case.

Table 4 summarizes our results concerning T_3. The same information is shown pictorially in Figure 15. This graph is intended to illustrate the general theory and has been distorted for the sake of clarity. (That is,

k	T_3	source
1	$\log n + 1$	Property A
$1 \leq k \leq e_n$	$\hat{T}_3 \leq T_3 \leq \hat{T}_3 + d_n$	Theorem 3
$e_n \leq k \leq s_n$	$\hat{T}_3 + d_n$	Theorem 4
$s_n \leq k \leq n$	T_1	Theorem 2

Table 4. T_3 as a function of k (for fixed n).

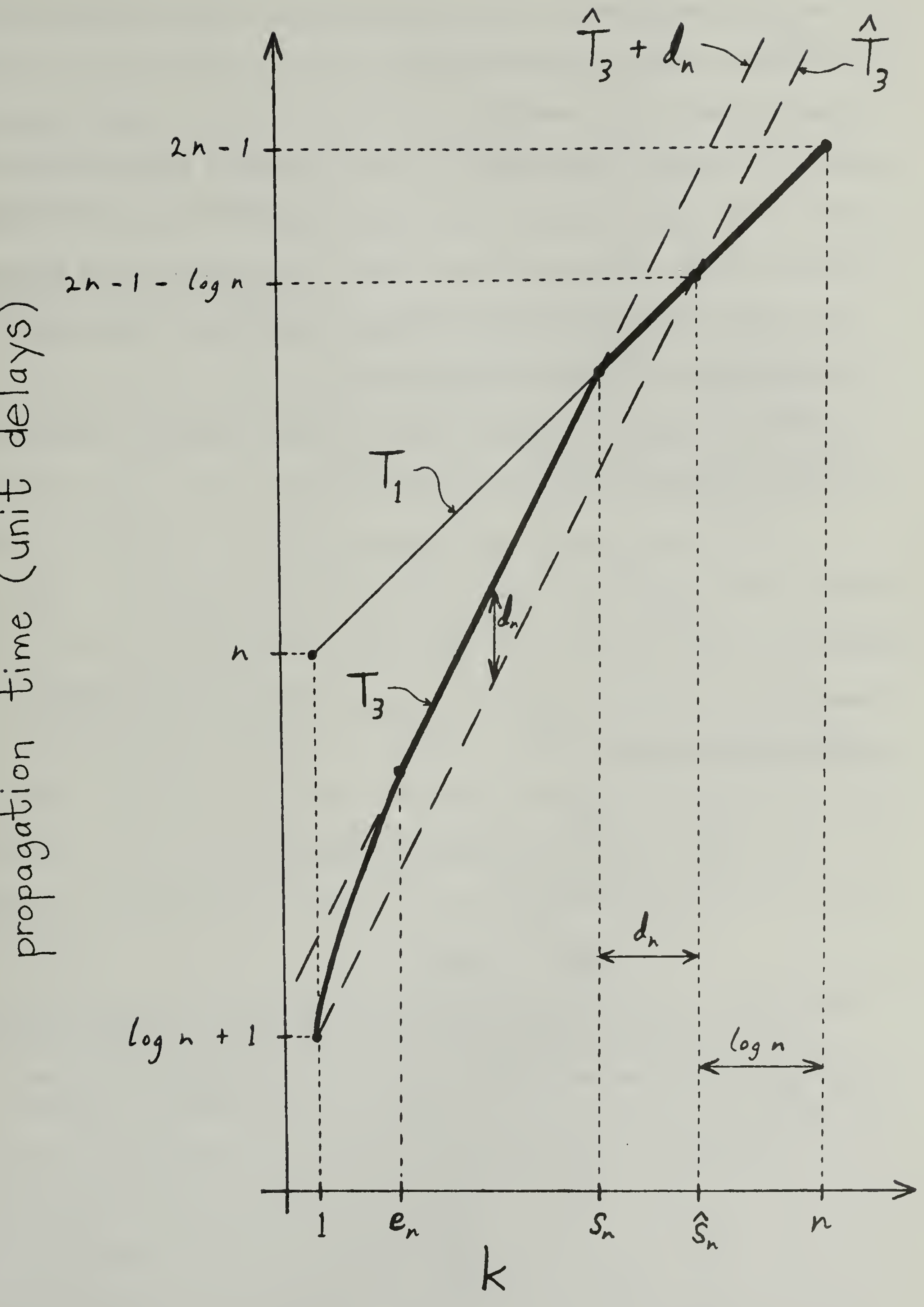

Figure 15. Speed comparison between Method 3 and Method 1.

the relative magnitudes of the various parameters are not as shown here.) We consider n fixed and have sketched T_1 and T_3 as functions of k, with k running from 1 to n. Note that $T_1 = n + k - 1$ is a straight line with slope 1. Since $\hat{T}_3 = \log n + 2k - 1$, both $\hat{T}_3$ and $\hat{T}_3 + d_n$ are lines of slope 2. (Of course, since k is an integer, none of these are in reality continuous lines. Moreover, we have drawn T_3 as a smooth curve between 1 and e_n only for the sake of the picture; actually, its exact behavior in this region is a matter of uncertainty.)

Taking $\hat{T}_3 + d_n$ as an approximation for T_3, we can compute the speed difference between Methods 1 and 3 as follows:

$$T_1 - T_3 \approx T_1 - (\hat{T}_3 + d_n) =$$

$$(n + k - 1) - (n - s_n + 2k - 1) = s_n - k$$

Assuming s_n is close to n and that k is not too small, we can also estimate the speedup:

$$\frac{T_1}{T_3} \approx \frac{T_1}{\hat{T}_3 + d_n} = \frac{n + k - 1}{n - s_n + 2k - 1} \approx$$

$$\frac{n + k}{2k} = \frac{1}{2}(1 + \frac{n}{k})$$

As a final note on Figure 15, we point out that T_3 crosses the line $\hat{T}_3$ at $\hat{s}_n$. This is because when $k = \hat{s}_n$, $T_3 = T_1$ (since $s_n \leq \hat{s}_n$), and thus we get:

$$n - \log n = k$$

$$n = \log n + k$$

$$n + k - 1 = \log n + 2k - 1$$

$$T_1 = \hat{T}_3$$

$$T_3 = \hat{T}_3$$

From this it follows that for all $k \leq \hat{s}_n$, $\hat{T}_3 \leq T_3 \leq \hat{T}_3 + d_n$. This is a slight extension of Theorem 3.

5.3. Some Empirical Results

The results of the last section are useful only if d_n and e_n are small compared to n. Unfortunately, general formulas for these quantities are not known. For that reason we wrote a computer program of the Method 3 algorithm which, for a given n, calculates T_3 for all values of k between 1 and that n. From this information, d_n and e_n are easily determined. The program was run for values of n up through 1024, in order to get an idea of the relative magnitudes of the various Method 3 parameters and to see how they vary with n.

Our results are presented in Table 5. Note that, as we had hoped, the crucial quantity d_n is indeed very small compared to n. In fact, as a fraction of n, d_n actually decreases as n approaches 1024. Although it is dangerous to extrapolate from this kind of data, the results are certainly encouraging.

Similarly, over the range investigated the parameter e_n stays fairly small; as a fraction of n it fluctuates within an interval bounded by 7 and 11 percent. This behavior is somewhat less promising than that of d_n, but it is good enough to indicate the usefulness of Theorem 4, especially since $\hat{s}_n$ seems to stay so close to n.

This brings us to the most impressive aspect of our table. Note that $\frac{s_n}{n}$ rises steadily as n increases. Beyond n = 64, this ratio is greater

n	4	8	16	32	64	128	256	512	1024
s_n	2	5	12	26	56	117	241	490	993
$\hat{s}_n$	2	5	12	27	58	121	248	503	1014
d_n	0	0	0	1	2	4	7	13	21
$s_n/\hat{s}_n$	1	1	1	.963	.966	.967	.972	.974	.979
e_n	(1)	(1)	(1)	3	5	11	20	52	108
s_n/n	.500	.625	.750	.813	.875	.914	.941	.957	.970
d_n/n	0	0	0	.031	.031	.031	.027	.025	.021
e_n/n	(.250)	(.125)	(.063)	.094	.078	.086	.078	.102	.105

Table 5. Method 3 parameter values.

than 90%. This is significant because $\frac{s_n}{n}$ represents that fraction of k's range over which Method 3 outperforms Method 1. If our data is at all representative, then indeed Method 3 is a significant improvement over the use of a rectangular array.

This conclusion is rendered even more compelling by Figures 16 through 19. These graphs show the complete results of our computer simulations for several values of n; T_1, T_2, $T_{2'}$, and T_3 have been plotted together for all values of k between 1 and n. (We have connected the plotted points so as to present continuous curves.) Thus from actual data we can directly compare the speeds of all the methods which we have discussed. In particular, we see that Method 3 is superior to Method 1 over nearly the entire range of k, the speed difference being especially drastic when $k << n$.

In these graphs the Method 3 bounds $\hat{T}_3$ and $\hat{T}_3 + d_n$ are shown as dashed lines. Note that $T_3 = \hat{T}_3 + d_n$ everywhere except in the extremes of k's range. Of course, for k close to n, $T_3 = T_1$. For k close to 0, the behavior of T_3 is somewhat erratic (though of course within the given bounds). We observe also that for all $k < \hat{s}_n = n - \log n$, T_3 is only slightly greater than $\hat{T}_3 = \log n + 2k - 1$.

5.4. Comparison with Method 2

In Figures 16 through 19 we also see that Method 3 is much faster than Method 2. In fact, this difference in performance becomes more marked for the larger values of n. For any given n, the speed difference increases with k. But note that even when k is so small that $T_2 < T_1$, we still have that $T_3 < T_2$.

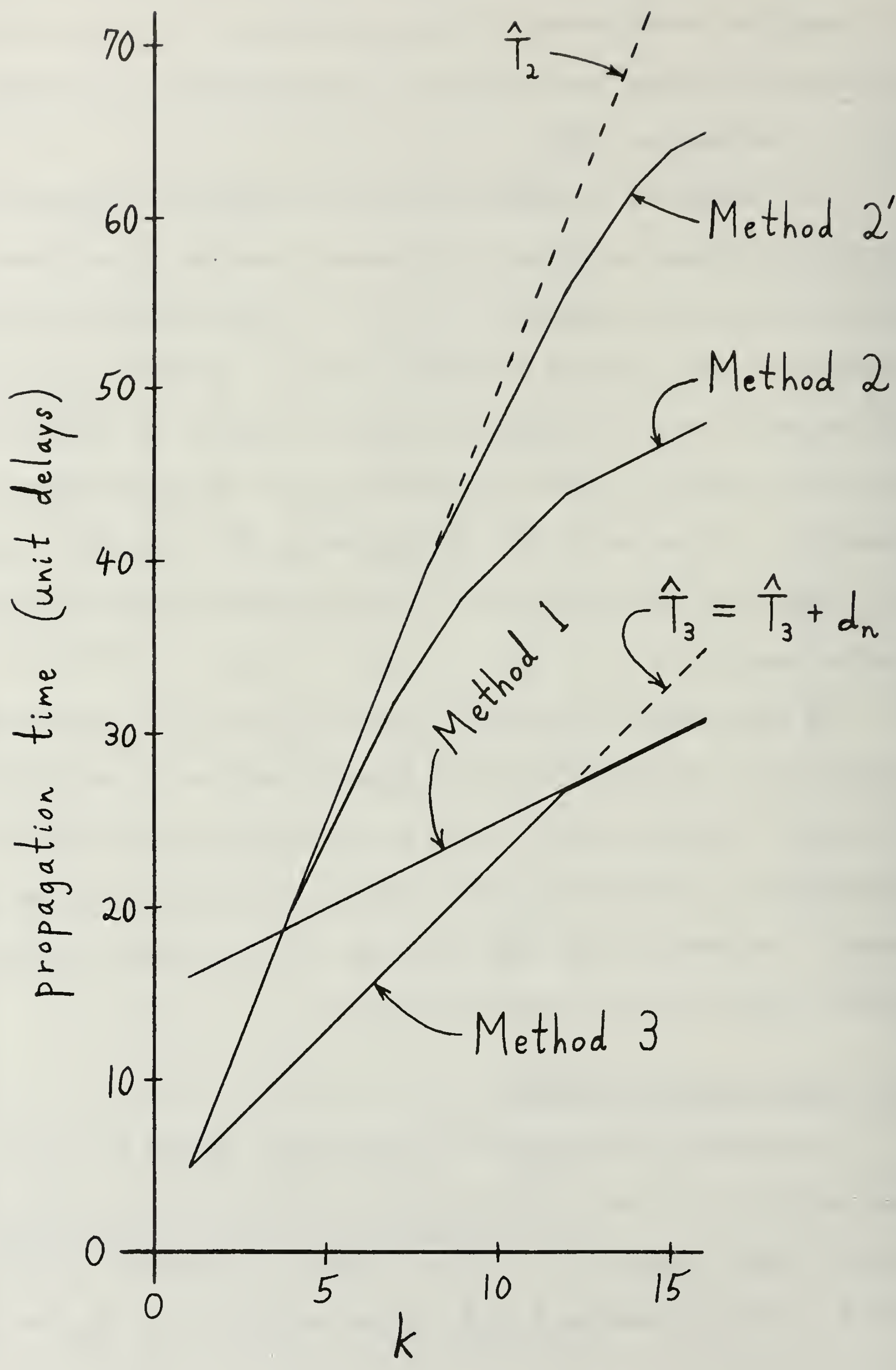

Figure 16. Speed comparison for n = 16.

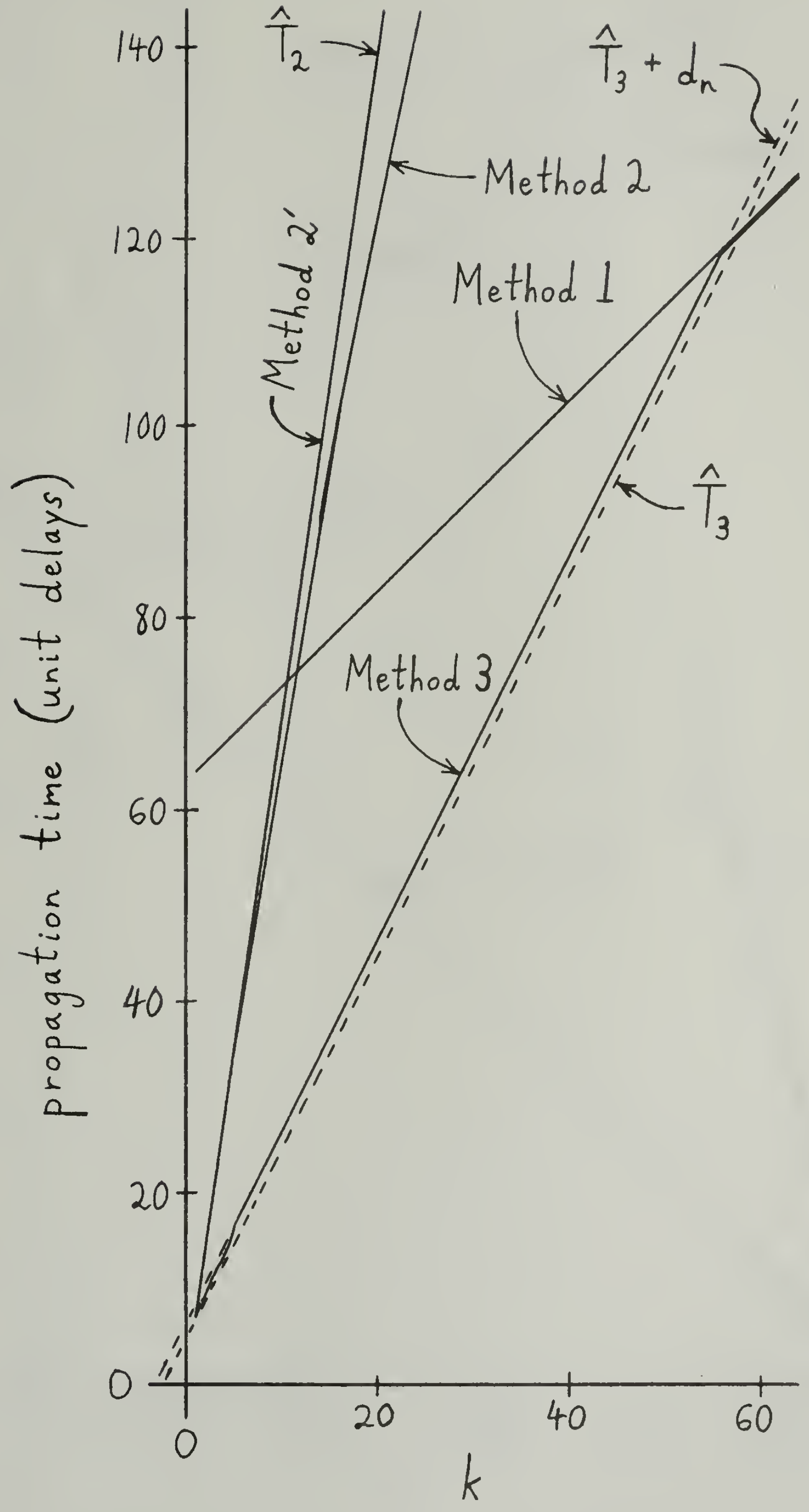

Figure 17. Speed comparison for n = 64.

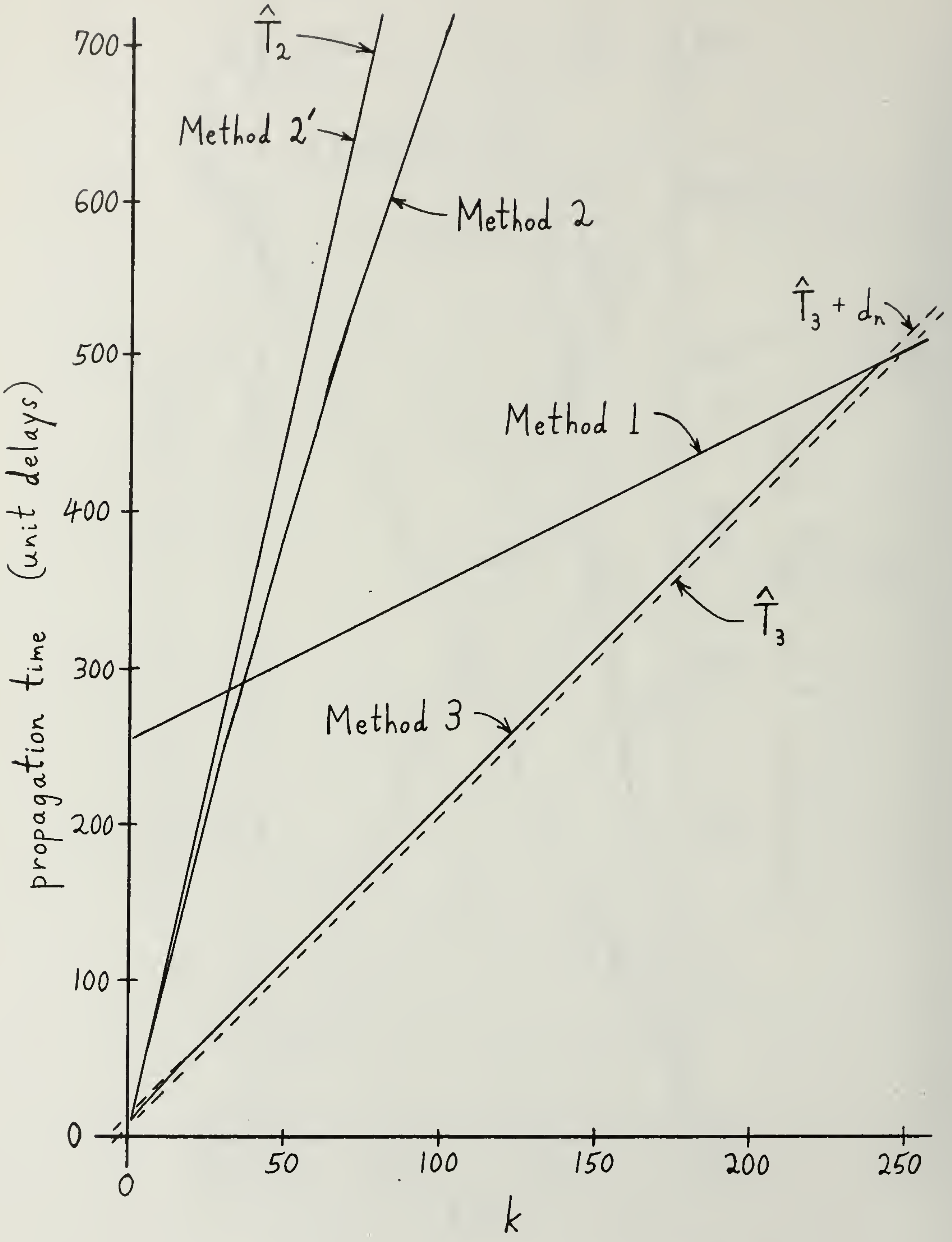

Figure 18. Speed comparison for n = 256.

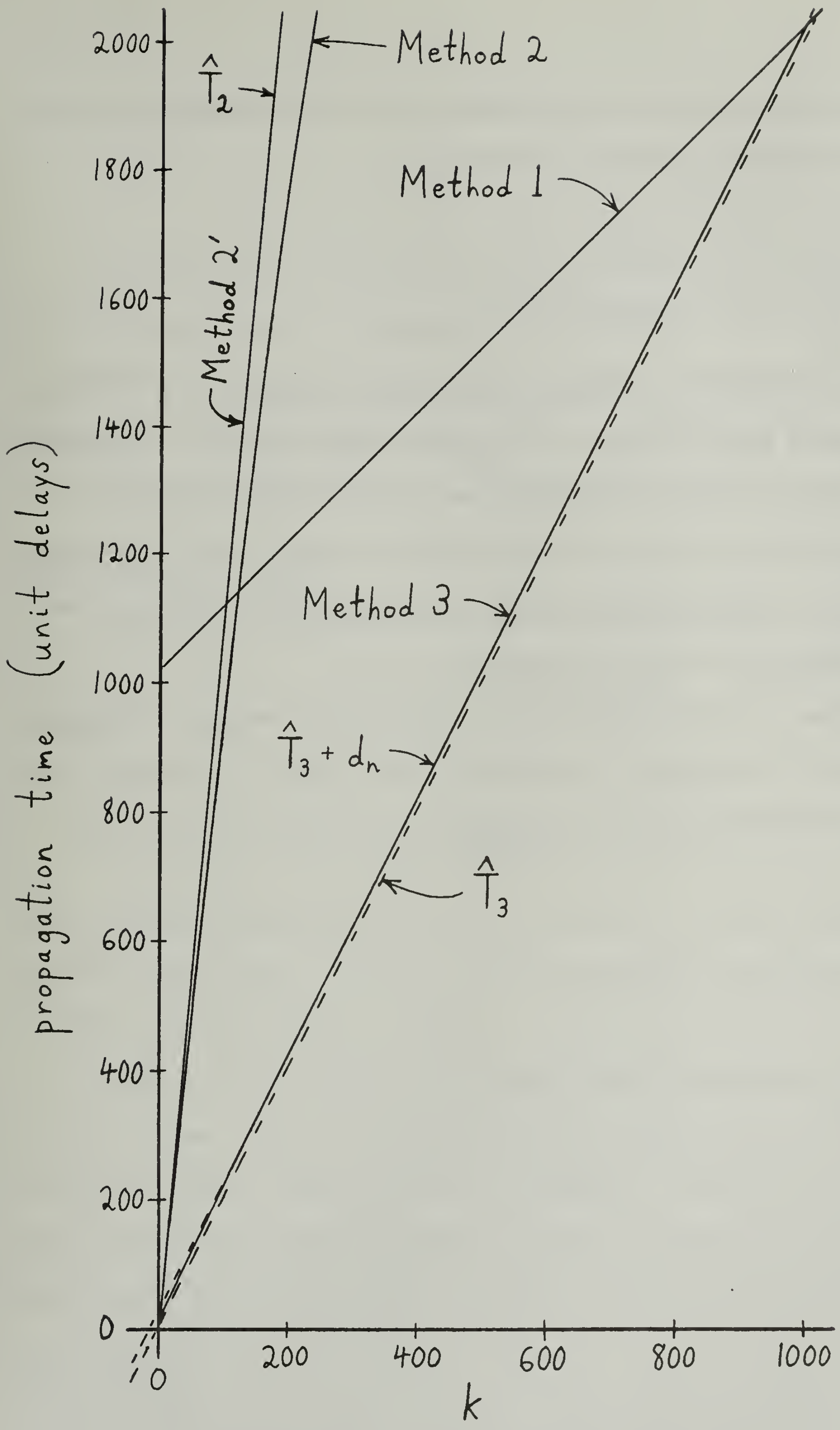

Figure 19. Speed comparison for n = 1024.

These results are not surprising in view of our approximations for the speeds of the two methods:

$$T_2 \approx \hat{T}_2 = k(\log n + 1)$$

$$T_3 \approx \hat{T}_3 = \log n + 2k - 1$$

With n fixed, $\hat{T}_2$ and $\hat{T}_3$ are straight lines which intersect at $k = 1$; but whereas $\hat{T}_2$ has a slope of $\log n + 1$, $\hat{T}_3$ has a slope of only 2. In reality, the speed difference between Methods 2 and 3 is not quite as great as this would indicate, since T_2 is slightly smaller than $\hat{T}_2$ and T_3 is slightly larger than $\hat{T}_3$. Nevertheless, the speedup is remarkably great, at least over the range of n for which we have data.

Of course, the discussion above applies equally well to a comparison between Method 2' and Method 3. In fact, since Method 2' is slower than Method 2, the speedup provided by Method 3 is in this case even larger.

CHAPTER 6

CONCLUSIONS

We present here a brief review of the results of this paper, comparing the various techniques we have introduced for transforming iterative arrays. The numbers n and k are the dimensions of the original rectangular (Method 1) network. T_i refers to the propagation time in unit delays of Method i. Recall that $T_1 = n + k - 1$.

6.1. Methods 2 and 2'

Method 2 has a delay time of order k log n. It is faster than Method 1 only when k is smaller than about $\frac{n}{\log n}$. In that case, the speedup of Method 2 over Method 1 is given by:

$$\frac{T_1}{T_2} \approx \frac{1}{k}\left(\frac{n}{\log n}\right)$$

(These approximations are valid for n at least as large as 1024 and probably much larger.) Method 2 uses more hardware than Method 1 and requires very high fan-out ratings.

Method 2' is a variation of Method 2 which necessitates much lower fan-out values. However, it uses even more hardware than Method 2. Method 2' is also slightly slower than Method 2. Both of these methods can be used to transform any Method 1 array, regardless of the cell operations # and * (see Figure 2), except that # must be associative.

6.2. Method 3

It appears that Method 3 can be applied only to arrays which satisfy fairly restrictive requirements. We conjecture necessary and sufficient conditions to be that * is associative, commutative, idempotent, and distributive over #, where # is associative and commutative. When Method 3 can be used, it is at least as fast as Method 1, regardless of the values of n and k. Although an exact formula for T_3 is unknown, much has been learned about the behavior of this function -- see Table 4 and Figure 15.

In practice, Method 3 is usually considerably faster than all the other methods discussed. Under certain assumptions, we find that Method 3 has a delay time of order log n + k, as well as speedups given by:

$$\frac{T_1}{T_3} \approx \frac{1}{2}\left(1 + \frac{n}{k}\right)$$

$$\frac{T_2}{T_3} \approx \frac{T_{2'}}{T_3} \approx \frac{1}{2} \log n$$

These results are borne out by empirical data gathered for values of n up to 1024. For larger n, we cannot be certain that Method 3 performs as well, although indications are promising.

The speed advantage of Method 3 over Method 1 is obtained at no hardware expense whatsoever. The Method 3 transformation is merely a rearrangement of the lines which interconnect the cells of the Method 1 array. Moreover, Method 3 entails no extra fan-out. The one disadvantage is that Method 3 produces an interconnection pattern which is very irregular.

6.3. Summary

Our results are summarized more concisely in Table 6. For Methods 2 and 2', "hardware required" means the number of #-modules plus the number of *-modules. (Otherwise, it means the number of #/*-modules.) For Methods 2 and 3, the speed approximations given in the table are valid for n up to at least 1024. However, the best way to appreciate the speed differences among all of these methods is to look at the graphs of actual data in Figures 16 through 19.

It is clear that where Method 3 can be applied it is vastly superior to the other methods we have considered. If Method 3 cannot be used, then Method 2 or Method 2' may be the best choice, but only if k is small compared to n. Otherwise, we shall have to be satisfied with the original Method 1 array. Future research should be directed toward developing transformation techniques with benefits at least as great as those of Methods 2, 2', and 3 but with less severe limitations.

	← general methods →			
	Method 1	Method 2	Method 2'	Method 3
<u>computation time</u> (unit delays)	$n + k - 1$	slightly less than $k(\log n + 1)$ (for $k \leq \sim \frac{n-1}{\log n}$)	$k(\log n + 1)$ (for $k \leq \frac{n}{2}$)	slightly more than $\log n + 2k - 1$ (for $k \leq n - \log n$)
<u>hardware required</u> (total number of modules)	nk	$nk(\frac{\log n}{2} + 1) + k$	$nk \log n + 2k$	nk
<u>maximum fan-out</u> (unit loads)	1	$\frac{n}{2} + 1$	$\log n$	1

Table 6. Comparison chart for (n,k) networks.

LIST OF REFERENCES

1. Hennie, F. C., Iterative Arrays of Logical Circuits, John Wiley and Sons, New York, 1961.

2. Minnick, R. C., "A Survey of Microcellular Research," Journal of the Association for Computing Machinery, Vol. 14, No. 2, pp. 203-241, April 1967.

3. Kautz, W. H., "Programmable Cellular Logic," Recent Developments in Switching Theory, Mukhopadhyay, ed., pp. 369-422, Academic Press, New York, 1971.

4. Akers, S. B., "A Rectangular Logic Array," IEEE Transactions on Computers, Vol. C-21, No. 8, pp. 848-857, August 1972.

5. Pezaris, S. D., "A 40-ns 17-Bit by 17-Bit Array Multiplier," IEEE Transactions on Computers, Vol. C-20, No. 4, pp. 442-447, April 1971.

6. Cappa, M. and V. C. Hamacher, "An Augmented Iterative Array for High-Speed Binary Division," IEEE Transactions on Computers, Vol. C-22, No. 2, pp. 172-175, February 1973.

7. Majithia, J. C., "Cellular Array for Extraction of Squares and Square Roots of Binary Numbers," IEEE Transactions on Computers, Vol. C-21, No. 9, pp. 1023-1024, September 1972.

8. Gajski, D. D., "Semigroups of Recurrences," High Speed Computer and Algorithm Organization, Kuck, et al., eds., pp. 179-183, Academic Press, New York, 1977.

9. Unger, S. H., "Tree Realizations of Iterative Circuits," IEEE Transactions on Computers, Vol. C-26, No.4, pp. 365-383, April 1977.

10. Kuck, D. J., The Structure of Computers and Computations, Vol. 1, John Wiley and Sons, New York, 1978.

11. Gajski, D. D., "Class Notes on Digital System Design Automation," Department of Computer Science, University of Illinois, Urbana, Illinois, 1978.

12. Ladner, R. E. and M. J. Fischer, "Parallel Prefix Computation," Technical Report No. 77-03-02, Department of Computer Science, University of Washington, Seattle, Washington, 1977.

13. McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill, New York, 1965.

14. Burton, D. M., Introduction to Modern Abstract Algebra, Addison-Wesley, Reading, Massachusetts, 1967.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1022	2.	3. Recipient's Accession No.

4. Title and Subtitle SPEEDING UP COMPUTATION IN TWO-DIMENSIONAL ITERATIVE ARRAYS	5. Report Date May, 1980 6.
7. Author(s) Peter Gerald Rose	8. Performing Organization Rept. No. UIUCDCS-R-80-1022
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801	10. Project/Task/Work Unit No. 11. Contract/Grant No. US NSF MCS80-01561
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.	13. Type of Report & Period Covered Master's Thesis 14.

15. Supplementary Notes

16. Abstracts

This thesis considers rectangular, unidirectional arrays containing identical cells of combinational logic. The goal was to find an algorithm for transforming any such array into a network which performs the same computation in less time. Two methods are presented. The first achieves speedup only if the length of the original array is much greater than its width; it also requires much additional hardware. The second merely rearranges the lines which interconnect the cells, yet it obtains substantial speedup for all but the most square arrays. However, the latter method can be applied only if the cell logic satisfies fairly restrictive conditions.

17. Key Words and Document Analysis. 17a. Descriptors

Combinational logic
Computation time
Iterative arrays

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 83
	20. Security Class (This Page UNCLASSIFIED	22. Price